UNIVERSIDADE FEDERAL DE SANTA CATARINA

Thiago Martendal Salvador

Geração de Malhas Adaptativas Usando Tabelas de Espalhamento

Florianópolis

2022

Thiago Martendal Salvador

Geração de Malhas Adaptativas Usando Tabelas de Espalhamento

Trabalho de Conclusão de Curso do Curso de Graduação em Ciência da Computação do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Álvaro Junio Pereira Franco, Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Salvador, Thiago Martendal
Geração de Malhas Adaptativas Usando Tabelas de Espalhamento / Thiago Martendal Salvador ; orientador, Álvaro Junio Pereira Franco, 2022.
76 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, , Programa de Pós-Graduação em , Florianópolis, 2022.

Inclui referências.

1. . 2. Tabelas de Espalhamento. 3. Malhas. 4. Estruturas de Dados. I. Franco, Álvaro Junio Pereira. II. Universidade Federal de Santa Catarina. Programa de Pós Graduação em . III. Título.

Thiago Martendal Salvador

Geração de Malhas Adaptativas Usando Tabelas de Espalhamento

Florianópolis, 5 de Agosto de 2022.

Prof. Álvaro Júnio Pereira Franco, Dr.
Orientador

Banca Examinadora:

Prof. Jean Everson Martina, Dr.
Avaliador

Prof. Alexandre Gonçalves Silva, Dr.
Avaliador

Catalina Maria Rúa Alvarez, Dr.
Avaliadora

Meu Pai, meu Amigo

Você partiu dessa vida, mas mesmo que sua presença não esteja mais aqui, você não nos deixou, e nunca deixará.

Dedicado ao meu Pai.

AGRADECIMENTOS

Agradeço a minha família, por me apoiar nos momentos mais difíceis.
Ao Álvaro, por me apoiar e me orientar neste trabalho.
Aos meus professores, por me mostrarem o gigante mundo da Ciência da Computação.
E a todos que fizeram parte dessa caminhada.
A Deus, por todas as oportunidades para adquirir conhecimento, por todo o meu desenvolvimento até aqui, e por todos os meus dias.

Agradeço ao meu Pai, por ter feito parte da minha vida.

RESUMO

Este trabalho apresenta a construção de tabelas de espalhamento com algoritmos numéricos e de estruturas de dados, para a representação e geração de malhas adaptativas com refinamento. As malhas são configuradas como uma rede de células, onde as células são o componente básico da malha. Cada célula da malha é armazenada na tabela de espalhamento seguindo uma função de espalhamento definida por uma operação numérica. Essa função mapeia um dado em determinada posição da tabela. A tabela de espalhamento deve oferecer operações básicas de interação com os dados em tempo O(1), pois uma malha pode gerar milhões de células, o que torna necessário a definição de operações eficientes para lidar com volumosas quantidades de dados. Por exemplo, a análise de temperatura em um ambiente, pode gerar malhas com uma grande quantidade de células.

Palavras-chave: Tabelas de Espalhamento. Malhas. Células.

ABSTRACT

This work presents the construction of hash tables with numerical algorithms and data structures, for the representation and generation of adaptive meshes with refinement. Meshes are configured as a network of cells, where cells are the basic component of the mesh. Each mesh cell is stored in the hash table following a hash function defined by a numerical operation. This function maps a data in a certain position in the table. The hash table must offer basic operations for interacting with the data in O(1) time, as a mesh can generate millions of cells, which makes it necessary to define efficient operations to deal with voluminous amounts of data. For example, the analysis of temperature in an environment can generate meshes with a large number of cells.

Keywords: Hash Tables. Meshes. Cells.

Lista de Figuras

Figura 1 - Uma malha de nível 2 sem índices....16
Figura 2 - Uma malha de nível 1 com 4 células....18
Figura 3 - Malha não uniforme de nível 3....19
Figura 4 - Uma Quad-tree armazenando células de uma malha....23
Figura 5 - Exemplo de tabela de espalhamento....24
Figura 6 - Um vetor de strings....26
Figura 7 - Uma malha de nível 1 mapeada em regiões....28
Figura 8 - Uma malha de nível 3 mapeada em regiões....30
Figura 9 - Uma tabela com elementos distribuídos pelo espalhamento universal....32
Figura 10 - Tabela de espalhamento em dois níveis....34
Figura 11 - Um exemplo de como os dados são dispostos com o espalhamento perfeito....45
Figura 12 - Uma possível aplicação de hipergrafo para armazenar as células vizinhas....46
Figura 13 - O hipergrafo para a malha da Figura 12....47
Figura 14 - Aplicação da busca de células vizinhas em uma malha uniforme de nível 1....51
Figura 15 - A busca de células vizinhas para uma malha uniforme de nível 2....52
Figura 16 - Uma malha não-uniforme de nível 3 onde a célula (4,3,3) é a célula de interesse. 53
Figura 17 - As células vizinhas de (4,3,3)....55
Figura 18 - Um exemplo da formação dos vizinhos no próximo nível....60
Figura 19 - Uma malha de nível 1 com refinamento na célula (0,1,1)....64
Figura 20 - As modificações realizadas na malha durante a remoção de uma célula....65
Figura 21 - Um exemplo de remoção de células com um processo recursivo....66

Lista de Quadros

Quadro 1 - O Pseudocódigo da estrutura de célula....................20
Quadro 2 - Valores ASCII dos caracteres utilizados nas strings....................27
Quadro 3 - Cálculo das chaves intermediárias e índices....................27
Quadro 4 - Valores obtidos aplicando-se as operações dos passos 1, 2 e 5....................29
Quadro 5 - Valores calculados para o exemplo seguindo o método de transposição de células 29
Quadro 6 - Operações aplicadas no exemplo acima....................32
Quadro 7 - A quantidade de células por nível....................34
Quadro 8 - Pseudocódigo para o cálculo da chave intermediária....................36
Quadro 9 - Pseudocódigo do método da divisão....................37
Quadro 10 - Um vetor de vetores com a estrutura *vector*....................38
Quadro 11 - Pseudocódigo do método de inserção....................39
Quadro 12 - Pseudocódigo do método de remoção....................40
Quadro 13 - Pseudocódigo do método de busca....................40
Quadro 14 - Pseudocódigo do método contém célula....................41
Quadro 15 - Pseudocódigo para o experimento de formação de células....................42
Quadro 16 - Pseudocódigo do experimento de geração aleatória de células....................43
Quadro 17 - O pseudocódigo para a expansão de uma célula....................49
Quadro 18 - O pseudocódigo para a retração de uma célula....................50
Quadro 19 - Resultados do primeiro passo de busca para a malha da Figura 16....................54
Quadro 20 - A função principal para a busca de células vizinhas....................55
Quadro 21 - O método *inserirVizinho*....................57
Quadro 22 - O método *procuraVizinhosProximoNivel*....................57
Quadro 23 - O método *procuraVizinhosNivelAnteior*....................58
Quadro 24 - O método *vizinhosProximoNivel*....................58
Quadro 25 - Posições para a formação de células de acordo com cada direção....................61
Quadro 26 - O pseudocódigo para a inserção de uma nova célula na malha....................63
Quadro 27 - Pseudocódigo do método *refinarCelula*....................64
Quadro 28 - Pseudocódigo do método *removerCelula*....................66
Quadro 29 - Pseudocódigo do método *removerVizinho*....................67
Quadro 30 - Pseudocódigo da função *removerCelulaProximoNivel*....................68

Sumário

1 INTRODUÇÃO....13
1.1 Motivação....13
1.2 Objetivos....14
1.2.1 Objetivo Geral....14
1.2.2 Objetivos Específicos....14
1.3 Definições....14
1.3.1 A Linguagem de Programação para o Trabalho Prático....14
1.3.2 Uma Estrutura de Dados Auxiliar....15
1.3.3 A Complexidade de Algumas Operações Básicas....15
1.3.4 Os Pseudocódigos....15
2 UMA MALHA....16
2.1 Representação de Uma Malha....17
3 UMA CÉLULA....18
3.1 Representação de uma Célula....20
4 ESTRUTURAS DE DADOS PARA O GERENCIAMENTO DE MALHAS. 21
4.1 Árvore K-Dimensional....21
4.2 Árvore de Alcance....22
4.3 Quad-tree....22
4.4 A Estrutura de Dados Escolhida....23
5 UMA TABELA DE ESPALHAMENTO....24
6 FUNÇÕES DE ESPALHAMENTO....26
6.1 A Chave Intermediária....26
6.2 O Método de Transposição de Células....27
6.3 Adaptação do Método de Transposição de Células....30
6.4 A Função de Espalhamento com o Método da Divisão....31
6.5 A Função de Espalhamento Universal....31
6.6 A Função de Espalhamento Perfeito....33
6.7 As Funções de Espalhamento Escolhidas....35
6.8 A Implementação das Funções Escolhidas....36
6.8.1 O Cálculo da Chave Intermediária....36
6.8.2 O Método da Divisão....37

7 A IMPLEMENTAÇÃO DA TABELA DE ESPALHAMENTO....................38
7.1 O Armazenamento dos Dados....................38
7.2 A Implementação das Operações....................38
7.2.1 Definições....................38
7.2.2 Inserção....................39
7.2.3 Remoção....................40
7.2.4 Busca....................40
7.2.5 Contém Célula....................41
8 EXPERIMENTOS E RESULTADOS....................42
8.1 Experimento - A Formação de Células....................42
8.2 Experimento - Geração Aleatória de Células....................43
8.3 Revisão do Funcionamento da Estratégia de Espalhamento....................44
9 AS VIZINHANÇAS....................46
9.1 A Primeira Ideia Para o Cálculo das Vizinhanças....................46
9.2 Operações Necessárias Para a Busca de Células Vizinhas....................48
9.2.1 A Expansão de uma Célula....................48
9.2.2 A Retração de uma Célula....................49
9.2.3 Observações....................50
9.3 A Estratégia de Busca de Células Vizinhas....................51
9.3.1 O Primeiro Passo de Busca....................51
9.3.2 O Segundo Passo de Busca....................53
9.3.3 Teste de Mesa....................53
9.3.4 O Pseudocódigo Para as Estratégias de Busca de Células Vizinhas....................55
9.3.5 Revisão Resumida da Ideia Geral Para a Busca de Vizinhanças....................61
10 OPERAÇÕES SOB AS CÉLULAS DA MALHA....................63
10.1 Espalhar....................63
10.2 Nova Célula....................63
10.3 Refinar Célula....................63
10.4 Remover Célula....................65
11 CONCLUSÃO....................69
REFERÊNCIAS....................71

1 INTRODUÇÃO

A proposta deste trabalho é desenvolver um software que gera malhas no plano. A manutenção de malhas exige operações eficientes sobre elementos da malha (células). O objetivo é inserir, remover e recuperar elementos da malha da forma mais eficiente possível. Serão analisadas estruturas de dados eficientes para esse propósito. Além das operações citadas anteriormente, será desenvolvida uma operação para encontrar células vizinhas de uma dada célula.

Com relação as operações de inserção, remoção e recuperação de elementos da malha, a proposta é utilizar uma tabela de espalhamento. E com relação a operação de encontrar células vizinhas, a proposta inicial era utilizar hipergrafos para o reconhecimento das vizinhanças. Porém, foram desenvolvidas formas mais simples para o mesmo fim.

As operações que devem demandar uma baixa complexidade referem-se as operações realizadas na tabela de espalhamento, porém, a malha em si também tem operações próprias, no entanto, estas podem demandar uma maior complexidade. O grande objetivo é garantir que a estrutura de armazenamento consiga garantir operações básicas em tempo O(1).

As área de Análise Numérica, Física, Engenharias e áreas afins são clientes do produto que este trabalho visa construir.

1.1 MOTIVAÇÃO

Como levantado no resumo do trabalho, o interesse em questão é aplicar algoritmos para geração de malhas adaptativas, que são usadas na resolução de problemas em áreas como Matemática Aplicada, Análise Numérica, Engenharias e Física. O intuito é construir mecanismos de alta eficiência que operem sob um grande consumo de memória computacional, já que a divisão de células em uma malha cresce exponencialmente. Cada célula é dividida em 4 novas células, e para uma malha uniforme de nível 10, o refinamento de células no nível 10, são geradas $4^{10} = 1048576$ células.

Para lidar com um grande número de células, tem-se a necessidade de usar uma estrutura de dados que consiga lidar com grandes quantidades de dados em um nível de complexidade aceitável para uma boa execução. Essa estrutura é a tabela de espalhamento (também conhecida como tabela de dispersão ou tabela *hash*). O uso das tabelas de espalhamento é interessante para o problema, pois sua construção propõe operações que conseguem lidar de forma eficaz com grandes quantidades de dados.

Dentro das tabelas de espalhamento, os dados são distribuídos através de uma função de espalhamento (também chamada de função de dispersão ou função *hash*), como mencionado anteriormente. As funções de espalhamento determinam a eficiência do espalhamento, portanto a escolha de uma boa função de espalhamento determina o bom funcionamento da estrutura de dados.

Portanto, um dos grandes objetivos deste trabalho é encontrar uma função que garanta o melhor espalhamento possível para as células da malha.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Construir estruturas para a representação de malhas e células, e uma tabela de espalhamento que tenha a aplicação de uma função de espalhamento que armazene os dados sem a ocorrência de colisões.

1.2.2 Objetivos Específicos

Com a proposta de se construir uma tabela de espalhamento que garanta o melhor resultado para o armazenamento de dados através de funções numéricas, também surge a necessidade de se ter operações de interação com os dados para esta tabela com complexidade O(1). Após a conclusão da tabela de espalhamento, será construída a malha, que se trata de uma estrutura intermediária que gerencia as operações de tabela e o tratamento de células em um todo.

1.3 DEFINIÇÕES

Neste tópico seguem algumas definições que foram aplicadas no trabalho.

1.3.1 A Linguagem de Programação para o Trabalho Prático

Para se programar as estruturas requeridas, foi utilizada a linguagem de programação C++, devido ao bom desempenho da linguagem e a praticidade com o desenvolvimento de estruturas de dados.

1.3.2 Uma Estrutura de Dados Auxiliar

Como as estruturas de dados finais propostas pelo trabalho têm um caráter mais avançado, as estruturas mais básicas não foram implementadas, sendo utilizadas bibliotecas da *Standard Template Library* (STL) do C++ para desempenhar as funções de armazenamento.

Para este trabalho, foi utilizada a estrutura *vector*, que pertence a biblioteca de mesmo nome, para representar vetores. O *vector* será utilizado para armazenar os dados da tabela de espalhamento, mas nenhuma operação específica da biblioteca será utilizada além de *operator[i]*. O método *operator[i]* permite realizar a inserção de um determinado dado na posição *i* do *vector*. Além desta operação, nenhuma outra foi necessária.

1.3.3 A Complexidade de Algumas Operações Básicas

Como descrito anteriormente, um dos objetivos do trabalho é a construção de uma tabela de espalhamento com as operações de inserção, remoção e busca de dados em tempo constante O(1). Portanto, os algoritmos e estruturas relacionadas a essas operações da tabela devem contar com operações O(1) para não se elevar a complexidade.

Para as operações aritméticas é considerado que qualquer operação consome uma unidade de tempo O(1).

A função *operator[i]* foi descrita anteriormente e tem uma complexidade de tempo constante O(1), como descrito na documentação da linguagem C++.

Para a malha, suas operações são construídas em uma classe que representa a malha, e como dito anteriormente, essas operações não necessitam ter uma complexidade baixa, até porque algumas operações da malha realizam muitas operações para concluir seus objetivos.

1.3.4 Os Pseudocódigos

Ao longo do trabalho são usados pseudocódigos de alto nível para descrever a implementação de métodos e funções importantes para o desenvolvimento do trabalho. Estes pseudocódigos são feitos com o intuito de informar a implementação de uma forma direta descrevendo operações e trechos de código que ocorrem em cada método.

2 UMA MALHA

Como descrito no resumo do trabalho, uma malha se trata de uma rede de células que podem se expandir até determinado nível. Este nível deve ser determinado antes da construção da malha e é conhecido como o nível máximo da malha. A malha tem esse comportamento para tornar possível a discretização de algum fenômeno ou problema, buscando sua solução ao mesmo tempo que pode ser possível criar uma representação gráfica.

A ideia de se dividir a malha em níveis tem como objetivo tornar acessível uma região em questão através dos identificadores das células, o que permite se trabalhar partes específicas da malha sem criar modificações em um domínio.

Desta forma, é possível operar problemas por partes que são descritas em diferentes regiões do domínio, contribuindo para uma solução final.

Toda a malha é descrita de um nível de 0 a n, com $n > 0$. O nível zero de uma malha se trata do nível base, que compõem a região total da malha. Uma malha pode ter um grande número de níveis, mas neste trabalho foram realizados testes de implementação considerando 12 como sendo o nível máximo da malha, pois na prática não serão tratadas malhas com níveis maiores.

A Figura 1 mostra um pequeno exemplo de uma malha de nível 2 com 16 células (pois $4^2 = 16$), sem índices em suas células.

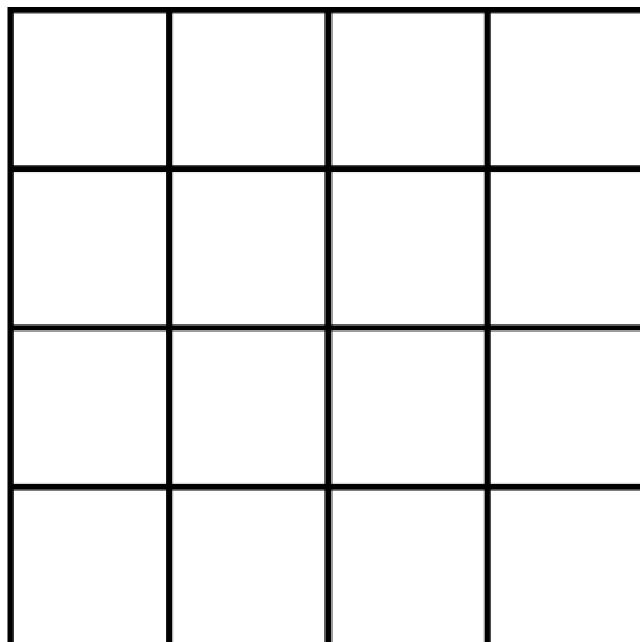


Figura 1 - Uma malha de nível 2 sem índices.

2.1 REPRESENTAÇÃO DE UMA MALHA

Uma malha pode ser representada com uma classe que contém uma estrutura de dados capacitada a armazenar as células (a tabela de espalhamento) e métodos que implementam as operações necessárias para a interação com as células.

As operações existentes na classe da malha não devem ser confundidas com as operações da tabela de espalhamento, pois a malha é pensada para se olhar as células como um todo, mesmo que estas possam ser operadas de forma individual. Já as operações da tabela se referem apenas as questões de armazenamento e resgate das células.

Apesar de ser interessante ter uma classe específica para a malha, é possível realizar operações em outros escopos de algum programa, basta se obter a tabela que armazena as células e então é possível programar novas operações além da classe especificada. E isso é importante para as futuras aplicações deste trabalho, pois os analistas numéricos podem construir suas próprias operações e métodos que a malha geral talvez não ofereça.

Para este trabalho, portanto, as operações necessárias a malha serão descritas na classe da malha.

3 UMA CÉLULA

Uma célula é uma componente elementar de uma malha. Uma célula é identificada por 3 valores inteiros: $c = (x, y, l)$, onde c identifica a célula, x é o índice da coordenada no eixo das abscissas, y é o índice da coordena no eixo das ordenadas, e l é o nível da célula, que é obtido a partir do processo de refinamento. Nenhum dos três valores de indexação são negativos e todos eles são números inteiros.

Os índices da tabela são obtidos seguindo o plano cartesiano 2D.

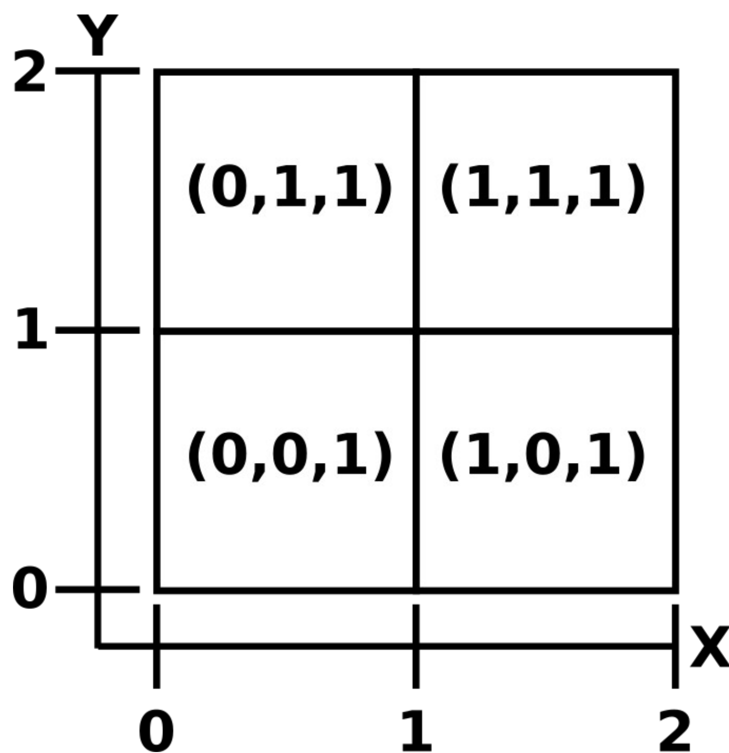


Figura 2 - Uma malha de nível 1 com 4 células.

A Figura 2 mostra o esquema de uma malha uniforme, onde todas as células se dividem de forma igual até atingir o nível em questão.

Como as células são classificadas em níveis, elas possuem espessura. Uma célula é dita pertencer ao nível mais fino se ela está no nível máximo permitido pela malha. Uma célula que está no primeiro nível, pertence ao nível mais grosso. Portanto, os níveis de cada célula representam sua espessura até o nível máximo da malha.

No caso de uma malha não-uniforme, pode-se ter a ocorrência de células de níveis diferentes, como mostra a Figura 3.

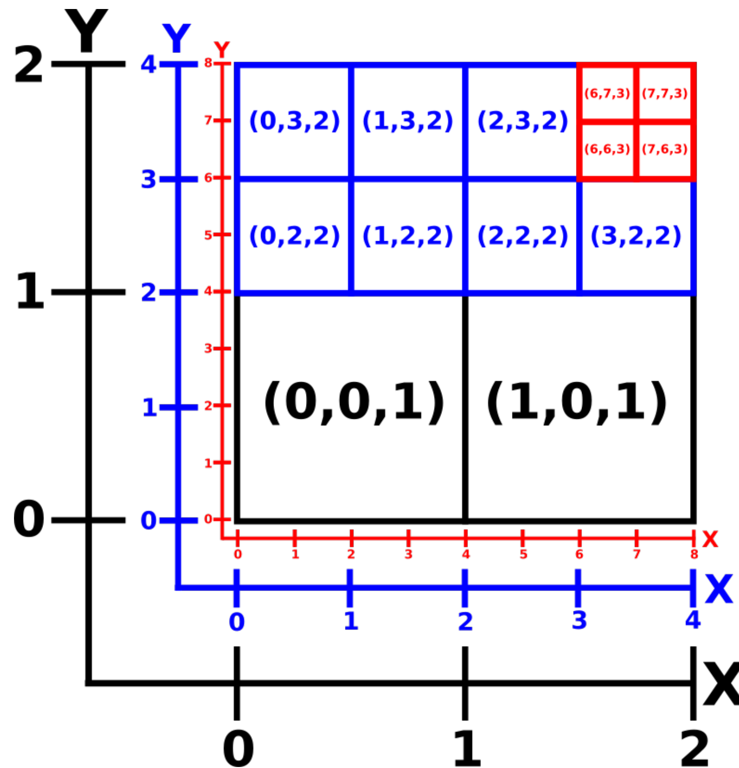


Figura 3 - Malha não-uniforme de nível 3.

Na Figura 3 foi representada uma malha não-uniforme de nível 3. Cada nível de células existente na malha foi representado em cores diferentes, para mostrar como pode ocorrer a disposição de células em uma malha não-uniforme. Para cada nível existente, a formação de índices das células segue um plano cartesiano específico, que será detalhado mais adiante. As células coloridas em preto pertencem ao nível mais grosso, e as células coloridas em vermelho pertencem ao nível mais fino. É importante ressaltar que o crescimento dos índices de células ocorre de acordo com o plano cartesiano em cada nível, logo, conforme os níveis avançam, o plano cartesiano é ajustado para atender o número total de células existentes naquele nível.

Cada célula em uma malha pode ser dividida em no máximo 4 células, logo, basta calcular 4^l para se descobrir o número total de células em um nível da malha, onde l representa o nível da célula. Este processo é denominado refinamento de célula.

Dada toda a descrição de como uma célula é composta, e como ocorre seu processo de refinamento, é definido que a única célula do nível 0 é (0, 0, 0), pois $4^0 = 1$. A célula do nível 0 cobre todo o domínio e representa o nível mais grosso da malha.

3.1 REPRESENTAÇÃO DE UMA CÉLULA

Neste trabalho, uma célula contém apenas seus valores de indexação, pois o foco não é fazer a aplicação das malhas, mas apenas a construção de estruturas que as representem.

Portanto, uma célula pode ser representada com uma estrutura, que se trata de um artifício computacional para agrupar dados (em algumas linguagens de programação uma estrutura é definida com o comando *struct*). Porém pode-se haver a necessidade de se conter mais dados além dos índices, e as células podem necessitar de operações específicas. Então, é possível representar uma célula com uma classe para estes casos, devido à facilidade de se construir um componente mais completo.

Para este trabalho, a célula será representada com uma estrutura que contém os 3 índices, e um parâmetro *nulo* para indicar se a célula está nula ou não, sendo esta representação suficiente para o desenvolvimento do trabalho. Também fica convencionado que os valores x, y e l são inicializados com 0, e o parâmetro *nulo* é inicializado com o valor *true* (verdadeiro), indicando que a célula está inicialmente nula. Quando as células forem inicializadas, os valores são alterados.

Quadro 1 - O Pseudocódigo da estrutura de célula

1	**estrutura** Célula
2	x = 0
3	y = 0
4	l = 0
5	nulo = verdadeiro
6	**fim estrutura**

O Quadro 1 apresenta o pseudocódigo da estrutura de célula descrita neste tópico.

4 ESTRUTURAS DE DADOS PARA O GERENCIAMENTO DE MALHAS

Nesta seção serão comentadas algumas estruturas de dados que podem ser utilizadas para o gerenciamento de malhas, mas que não foram aplicadas ao trabalho.

4.1 ÁRVORE K-DIMENSIONAL

Uma Árvore de Busca Binária Multidimensional (também conhecida como Árvore *K-Dimensional* ou Árvore *K-D*) é uma estrutura de dados baseada em árvore, que contém um espaço *k*-dimensional, onde *k* é a dimensionalidade do espaço de busca.

Os nós de uma Árvore *K-D* são organizados como pontos *k*-dimensionais no plano da dimensão definida, por isso, a quantidade de coordenadas em cada ponto varia de acordo com *k*. Por exemplo, se uma árvore tem dimensão 2, é uma árvore 2D e cada nó é formado por um ponto $p = (x,y)$ de chaves no plano, se tem dimensão 3, é uma árvore 3D e cada nó é formado por um ponto $p = (x,y,z)$, e assim por diante.

Uma Árvore *K-D* para um conjunto *P* de *n* pontos usa armazenamento O(n) e pode ser construída em tempo O(n log n). Porém, uma árvore com dimensionalidade 2 é suficiente para armazenar as células de uma malha, devido ao fato das malhas corresponderem ao plano cartesiano 2D, o que facilita o mapeamento das regiões da malha. O tempo de consulta em uma árvore 2D é $O(\sqrt{n}+k)$, onde *n* é o número de pontos do conjunto *P* e *k* é o número de pontos relatados.

No entanto, existe um problema com o uso de uma árvore 2D, pois uma malha pode ter um mesmo ponto em níveis diferentes, como é o caso de uma malha não uniforme que tenha as células (1,0,1) e (1,0,2), que tem o ponto em comum (1,0) mas pertencem a níveis diferentes. A malha da Figura 4 apresenta esta situação, e caso fosse ser armazenada em uma Árvore *K-D*, existiria a colisão de dados no ponto (1,0). Para se resolver este problema, é anexada uma árvore binária secundária aos nós da Árvore *K-D* para se armazenar as células. Isto é possível pois a árvore tem mais de uma dimensão. Então, o ponto (1,0) que representa um nó da árvore, tera uma árvore secundária que contém as células (1,0,1) e (1,0,2). Também pode ser usada uma árvore 3D, que admite pontos com três coordenadas, onde a terceira coordenada representa o nível da malha. Porém, as complexidades das operações da árvore aumentariam. Estas são possíveis soluções aplicando-se Árvores *K-D*, mas outras soluções podem ser propostas e aplicadas.

A árvore contém as operações básicas de inserção e remoção que são essenciais para a manipulação dos dados, no entanto, algumas complexidades de execução das operações são bem elevadas (como as apresentadas anteriormente nesta sessão). Isso faz a Árvore *K-D* não ser uma estrutura de interesse para este trabalho.

4.2 ÁRVORE DE ALCANCE

Uma Árvore de Alcance é uma estrutura de dados de árvore ordenada para conter uma lista de pontos. A Árvore de Alcance é uma estrutura apta a lidar com *k* dimensões, sendo uma alternativa a Árvore *K-D*. Para *k* dimensões, existem *k* níveis na árvore, onde cada dimensão representa um nível.

Para a aplicação na geração e processamento de malhas neste trabalho, é suficiente a utilização de uma Árvore de Alcance com duas dimensões, onde um ponto $p = (x,y)$ pode representar uma célula no plano cartesiano. Para a árvore de com duas dimensões, existiram dois níveis. Os nós do primeiro nível da árvore podem ser usados para representar os níveis da malha, e os nós do segundo nível da árvore podem ter árvores secundárias anexadas para armazenar as células. Os níveis da árvore e da malha não devem ser confundidos, o nível da malha diz respeito a formação das células, e o nível da árvore corresponde com a dimensão da mesma, onde a árvore principal é a árvore de primeiro nível, e nos demais níveis são alocadas as árvores associadas, sendo esta solução muito parecida com a solução aplicada para Árvore *K-D*. Esta é uma possível solução para a aplicação de Árvores de Alcance na geração de malhas, mas esta estrutura pode ser explorada de outras formas.

O tempo de consulta em uma Árvore de Alcance é O($\log^2(n+k)$), que é melhor que o tempo de consulta em uma Árvore *K-D*, porém, o tempo de armazenamento passa a ser O($n \log n$), e em Árvores *K-D* o tempo de armazenamento era O(*n*). A variável *n* é o total de elementos na árvore, e a variável *k* é o total de dimensões/níveis da árvore.

Devido a estas complexidades, a Árvore de Alcance também não foi aplicada neste trabalho.

4.3 QUAD-TREE

Uma *Quad-tree* é uma estrutura de dados organizada em árvore, onde cada nó sempre gera 4 nós filhos. Cada nó da *Quad-tree* corresponde a uma região. Esta região é subdividida em 4 novas regiões, gerando um novo nível na árvore. Esta subdivisão ocorre de forma recursiva, perdurando até se atingir um nível desejado.

Devido a forma como os nós se subdividem em uma *Quad-tree*, é fácil observar o processo de refinamento das células de uma malha, pois uma única célula também é dividida em 4 novas células. Então, uma *Quad-tree* é uma estrutura interessante para o armazenamento de células, pois conforme a malha é refinada, a estrutura se adapta para receber as novas células, gerando um novo nível na *Quad-tree* a partir dos níveis anteriores.

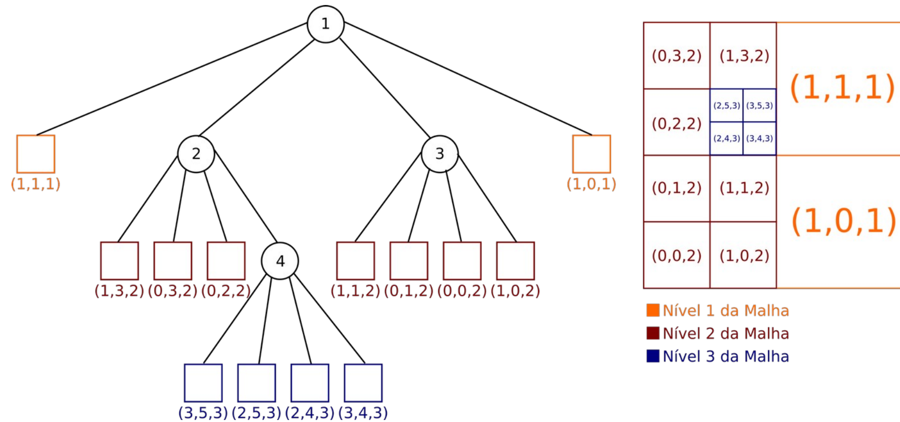


Figura 4 - Uma Quad-tree armazenando células de uma malha.

Fonte: Adaptação de "Computational Geometry – Algorithms and Applications Third Edition" – página 310.

Na Figura 4, é feita a representação de uma Quad-tree armazenando as células de uma malha não uniforme de nível 3. A malha foi colorida para destacar os níveis existentes, e cada nó da árvore que corresponde a uma célula da malha tem a mesma cor.

Como dito anteriormente, a árvore sempre gera 4 novos nós a partir de um nó anterior, onde cada nó corresponde a uma região. Portanto, foi necessário expandir os nós da árvore de forma que correspondesse com a malha gerada. Nas células onde houve um processo de refinamento dividindo a célula em 4 novas células, foi feita uma expansão no nó correspondente, onde 4 novos nós são adicionados a árvore para armazenar as novas células obtidas do processo de refinamento. Mas este processo deve ser feito com cautela para não se construir uma árvore com um excesso de nós filhos, o que gera células além do desejado. A árvore construída neste exemplo tem profundidade 3 e armazena as 13 células geradas na malha.

Uma *Quad-tree* de profundidade d que armazena um conjunto de n elementos possui nós $O((d+1) \times n)$ e pode ser construída em tempo $O((d+1) \times n)$.

Embora a estratégia de divisão dos nós e alocação dos elementos seja bastante interessante devido a similaridade com o processo de refinamento das células da malha, a estrutura *Quad-tree* também não será aplicada neste trabalho, devido os tempos apresentados

acima, mas é uma estrutura alternativa que pode ser bem aplicada em outros projetos envolvendo a gerência de malhas.

4.4 A ESTRUTURA DE DADOS ESCOLHIDA

Como dito anteriormente, a estrutura de dados escolhida para o armazenamento das células neste trabalho é a Tabela de Espalhamento, pois suas operações tem complexidade O(1), que é um dos requisitos do trabalho.

5 UMA TABELA DE ESPALHAMENTO

A tabela de espalhamento é uma estrutura de dados que realiza o mapeamento de dados para uma determina posição através de um índice, que é calculado a partir de uma função de espalhamento. O dado quando atribuído a um índice é chamado de dado satélite. A função de espalhamento tem por objetivo calcular um índice através de parâmetros fornecidos para algum método numérico que realizará o cálculo. Após este processo, é retornado o índice em questão que representa o local na tabela de espalhamento onde o dado será alocado. Esta configuração dá a ideia da existência de um par entre índice e dado satélite, como ocorre em estruturas como mapas e dicionários, porém, em tabelas de espalhamento a ocorrência de pares não é totalmente concreta, pois dados diferentes podem ser alocados para um mesmo índice, gerando uma colisão.

As colisões se tratam da disputa de dois ou mais dados satélites diferentes por um mesmo local na tabela de espalhamento. Essa disputa ocorre quando o método numérico escolhido para gerar os índices, gera índices repetidos para dados diferentes, mesmo que estes dados não tenham parâmetros repetidos. Portanto, deve-se avaliar maneiras de se lidar com a ocorrência de colisões no processo de espalhamento. A forma mais tradicional para impedir colisões é utilizar listas encadeadas nas posições da tabela de espalhamento, assim, quando uma colisão ocorrer, os dados serão armazenados na lista que está na posição selecionada. Esse método para a solução de colisões é chamado de encadeamento. Embora este processo evite a perda de dados, não foi aplicado neste trabalho, pois não permite a construção de operações com baixa complexidade já que para encontrar determinado valor na lista seria necessário fazer um percurso sobre a mesma.

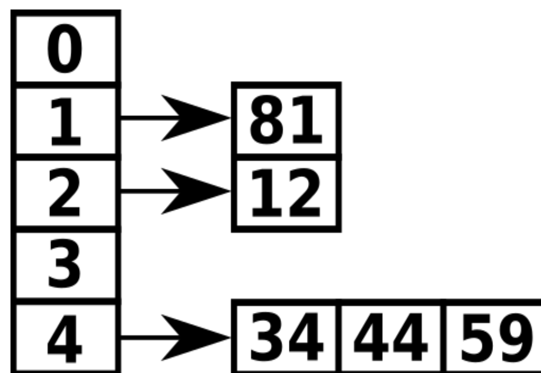


Figura 5 - Exemplo de tabela de espalhamento.

A Figura 5 mostra uma tabela de espalhamento com capacidade para se inserir até 5 elementos, tendo ocorrido colisões na posição 4 da tabela.

É usado um vetor ou uma lista que dispõem de uma lista encadeada em cada posição para se resolver o problema das colisões. Caso não fosse utilizado nenhum método para tratar as colisões, alguns dados seriam perdidos. Neste exemplo, se as colisões não fossem tratadas, o valor 34 seria substituído pelo valor 44, que seria substituído pelo valor 59.

6 FUNÇÕES DE ESPALHAMENTO

Uma função de espalhamento – também conhecida como função de dispersão ou função *hash* – se trata de uma função que mapeia uma chave intermediária para uma determinada posição da tabela de espalhamento, como mencionado anteriormente. Essa função é calculada através de uma operação numérica que utiliza determinados parâmetros incluindo a chave de uma célula. Abaixo serão apresentadas algumas funções de espalhamento que foram consideradas para implementações anteriores da tabela de espalhamento, juntamente com a função de espalhamento escolhida para a implementação final.

6.1 A CHAVE INTERMEDIÁRIA

Para se calcular o espalhamento com as funções numéricas, é necessário ter uma chave que representa o valor a ser inserido na tabela de espalhamento. Para uma tabela que armazena valores inteiros, a chave intermediária pode ser o próprio valor a ser inserido, já que se trata de um inteiro. Porém, nem sempre existe a conveniência de se armazenar apenas valores numéricos na tabela. Para o caso da inserção de valores não-numéricos, o dado a ser inserido pode ser acompanhado de um parâmetro que indica qual é o valor da chave intermediária, ou a chave pode ser obtida a partir de operações numéricas que são feitas utilizando parâmetros ou propriedades existentes no dado a ser inserido.

A Figura 6 apresenta um exemplo de como funciona o cálculo de uma chave intermediária para a inserção de strings em um vetor:

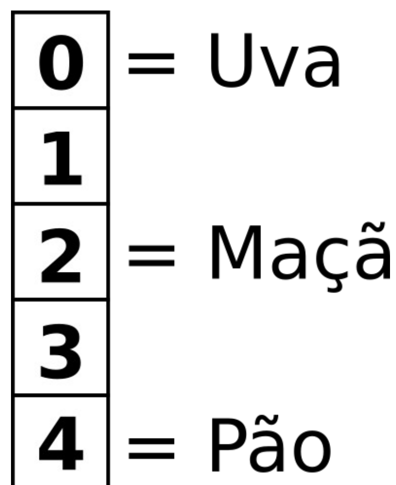


Figura 6 - Um vetor de strings.

Para o vetor acima, foi feito o cálculo da chave intermediária somando-se os valores ASCII dos caracteres em cada string que foi adicionada. Para se determinar o índice do vetor em que uma string é adicionada, foi feito o cálculo do resto da divisão da chave intermediária pelo tamanho do vetor, que é 5. Abaixo seguem as operações aplicadas no exemplo.

Fórmulas aplicadas no exemplo:

$$\text{Chave Intermediária} = \sum \text{Valor ASCII de cada caractere}$$

$$\text{Índice} = \text{Chave Intermediária} \ \% \ \text{Tamanho do Vetor}$$

Valores do exemplo:

Quadro 2 - Valores ASCII dos caracteres utilizados nas strings

U	M	P	a	o	ç	ã	v
85	77	80	97	111	135	198	118

Quadro 3 - Cálculo das chaves intermediárias e índices

	Chave Intermediária	**Índice**
Uva	85 + 118 + 97 = 300	300 % 5 = 0
Maçã	77 + 97 + 135 + 198 = 507	507 % 5 = 2
Pão	80 + 198 + 111 = 389	389 % 5 = 4

Este exemplo foi proposto para mostrar como pode ocorrer o cálculo de uma chave intermediária. A chave pode ser obtida utilizando-se parâmetros dos dados inseridos, por exemplo, no caso de inserção de strings, podem ser obtidos o tamanho da string ou os valores ASCII de cada caractere, o que permite a realização de alguns cálculos para se obter uma chave que atenda o problema em questão.

Para este trabalho, foi utilizada uma adaptação do método de transposição de células (subseção 6.2) para se calcular uma chave intermediária para as células, que será apresentado a seguir.

6.2 O MÉTODO DE TRANSPOSIÇÃO DE CÉLULAS

O método da transposição de células foi desenvolvido para realizar o mapeamento de células em regiões da malha. Esta operação é feita com algumas operações numéricas sobre os índices da célula.

Dada uma célula $c = (x, y, l)$, onde l é o nível da célula, para se calcular a transposição de células, seguem os passos:

1. Calcular a diferença de níveis:
   $\text{Diferença de nível} = \text{Nível máximo da malha} - \text{Nível da célula (l)}$, onde o *Nível máximo da malha* é um parâmetro da malha.
2. Calcular uma potência de 2 com a diferença de nível:
   $\text{Potência} = 2^{\text{Diferença de nível}}$
3. Encontrar a abscissa correspondente a x no nível mais fino:
   $\text{Abscissa} = x * \text{Potência}$
4. Encontrar a ordenada correspondente a y no nível mais fino:
   $\text{Ordenada} = y * \text{Potência}$
5. Neste passo, é obtido o número total de colunas da tabela de espalhamento:
   $\text{Número de colunas} = 2^{\text{Nível máximo da malha}}$
6. Por fim, é obtida uma chave operando os valores de abscissa e ordenada com o número de colunas do nível máximo:
   $\text{Região da malha} = \text{Número de colunas} * \text{Ordenada} + \text{Abscissa}$

A Figura 7 mostra um pequeno exemplo da aplicação da transposição de células para o reconhecimento das regiões das células.

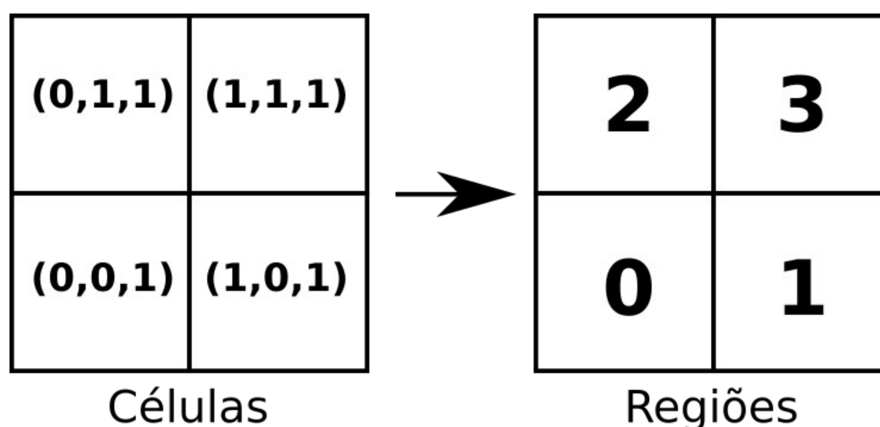


Figura 7 - Uma malha de nível 1 mapeada em regiões.

Para realizar o mapeamento das células nas regiões mostradas na Figura 7, foram aplicados os seguintes passos:

Quadro 4 - Valores obtidos aplicando-se as operações dos passos 1, 2 e 5

Nível máximo da malha	1
Nível das células	1
Diferença de níveis	0
Potência da diferença de nível	1
Número de colunas	2

No Quadro 4 estão descritos os valores obtidos aplicando-se as operações dos passos 1, 2 e 5 para o exemplo ilustrado na figura 6. Como explicado nos passos do método, o nível máximo da malha e o nível da células são parâmetros previamente definidos, e não são alterados.

No Quadro 5 segue a aplicação dos passos 3, 4 e 6 para se definir as regiões de cada célula:

Quadro 5 - Valores calculados para o exemplo seguindo o método de transposição de células

(0, 0, 1)		**(1, 0, 1)**		**(0, 1, 1)**		**(1, 1, 1)**	
Abscissa	0	Abscissa	1	Abscissa	0	Abscissa	1
Ordenada	0	Ordenada	0	Ordenada	1	Ordenada	1
Região	0	Região	1	Região	2	Região	3

Como todas as operações foram descritas anteriormente, neste exemplo foram apresentado apenas os resultados finais para cada uma das células.

Embora este exemplo seja simples, a aplicação do método de transposição de células ocorre sempre da mesma forma para todas as células em todos os níveis.

A Figura 8 mostra a transposição das células de uma malha de nível 3. Devido à quantidade de células, as operações foram omitidas, mas basta aplicar o método em cada uma das células que o resultado será igual.

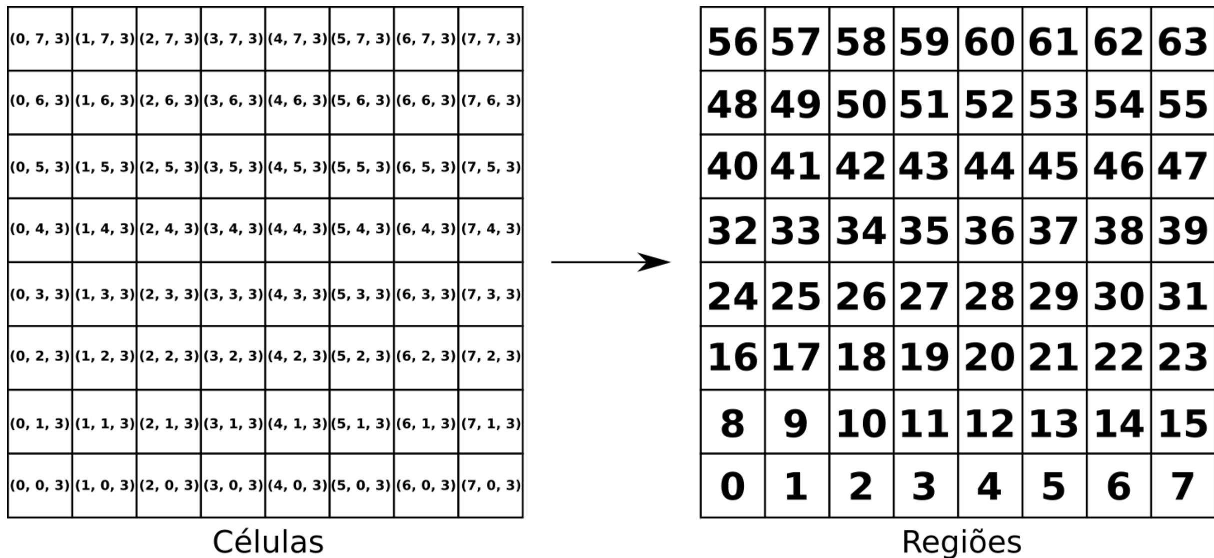


Figura 8 - Uma malha de nível 3 mapeada em regiões.

Também existem regiões em níveis diferentes dos níveis onde existem células. Por exemplo, as células (6,0,3), (7,0,3), (6,1,3) e (7,1,3) estão alocadas nas regiões 6, 7, 14 e 15, respectivamente. Porém, estas mesmas células estão dentro da região que é ocupada pela célula (3,0,2), que é a célula do nível 2 que está alocada neste mesmo lugar, mesmo não existindo na malha da Figura 8. Ao se calcular qual seria o valor da região de (3,0,2) para a malha da Figura 8, o resultado obtido é 6, portanto, as regiões 6, 7, 14 e 15 do nível 3 estão dentro da região 6 do nível 2, o que mostra que mesmo que a célula (3,0,2) não exista nessa malha, a sua região existe pois ela é compreendida pelas células (6,0,3), (7,0,3), (6,1,3) e (7,1,3).

6.3 ADAPTAÇÃO DO MÉTODO DE TRANSPOSIÇÃO DE CÉLULAS

A partir do cálculo apresentado, foi feita uma redução no número de operações, onde não é mais necessário encontrar a abscissa e a ordenada. Ao invés de se utilizar estes valores, basta aplicar aplicar diretamente os valores x e y da célula no cálculo da chave.

Portanto, seguem as operações:

1. Número total de colunas no nível da célula $= 2^{\text{Nível da célula}}$
2. Chave $=$ (Número total de colunas no nível da célula $*$ y) $+$ x

A chave obtida a partir deste cálculo pode ser utilizada como chave intermediária para ser aplicada em uma função de espalhamento, mas também pode ser diretamente aplicada

como uma posição para armazenar um elemento na tabela de espalhamento, sem a necessidade de se usar uma função de espalhamento mais específica, ou seja, um endereçamento direto com a chave intermediária. Isso é possível devido à forma como o espalhamento perfeito foi aplicado na tabela de espalhamento, que será discutido nas próximas seções.

6.4 A FUNÇÃO DE ESPALHAMENTO COM O MÉTODO DA DIVISÃO

O espalhamento com o método da divisão se trata de um cálculo extremamente simples para se descobrir o índice onde um dado será inserido na tabela de espalhamento.

Para se obter o índice do dado basta apenas calcular o resto da divisão entre a chave intermediária e o tamanho que foi alocado para a tabela de espalhamento.

Portanto:

$h(k) = k \% m$

Onde *k* é a chave intermediária, *m* é o tamanho da tabela, e *h(k)* é a aplicação da função de espalhamento. No tópico que apresentou a necessidade de se calcular a chave intermediária, foi dado um exemplo que utiliza o método da divisão para calcular o índice dos elementos.

6.5 A FUNÇÃO DE ESPALHAMENTO UNIVERSAL

O espalhamento universal se trata de um método que realiza uma escolha aleatória de uma função de espalhamento para se aplicar a tabela de espalhamento.

A escolha de uma função de espalhamento no método universal pode ser feita atribuindo-se parâmetros de seleção que determinam a escolha perfeita para ser aplicada. Esses parâmetros nada mais são que escolhas de projeto para se programar um seletor de funções. Este seletor pode ser puramente randômico, ou pode seguir diretivas programadas que influenciem na escolha que o espalhamento universal faz.

Mas além dos métodos de escolha, também é possível construir uma classe de cálculo de espalhamento própria seguindo a fórmula:

$h(k) = (((a \times k) + b) \% p) \% m$

Onde:

k = valor da chave

$a \in \{1, ..., p\}$ e $b \in \{0, ..., p\}$ onde p é um primo maior que o tamanho da tabela

m = tamanho da tabela de espalhamento

Os valores a e b são escolhidos aleatoriamente no intervalo. Dessa forma, é possível calcular uma função de espalhamento própria com espalhamento universal.

Tendo esclarecido o método, é possível construir uma classe de espalhamento universal que conta com o método descrito acima, além de outras funções de espalhamento que podem auxiliar no projeto.

Na Figura 9 segue um exemplo de uma tabela de espalhamento que utiliza o espalhamento universal.

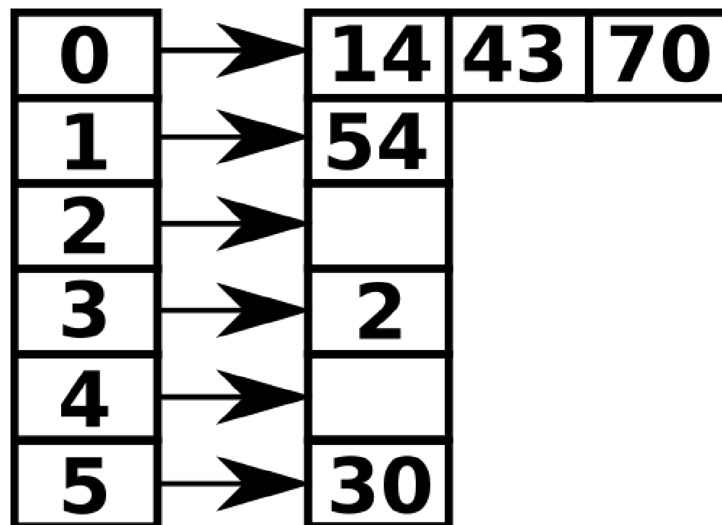


Figura 9 - Uma tabela com elementos distribuídos pelo espalhamento universal.

Na tabela da Figura 9 foi inserido o conjunto de inteiros 2, 14, 30, 43, 54, 70 tendo resultado nas seguintes operações:

Quadro 6 - Operações aplicadas no exemplo acima

$(((5 \times 2) + 41) \bmod 101) \bmod 6 = 3$
$(((98 \times 14) + 13) \bmod 101) \bmod 6 = 0$
$(((55 \times 30) + 49) \bmod 101) \bmod 6 = 5$
$(((10 \times 43) + 28) \bmod 101) \bmod 6 = 0$
$(((85 \times 54) + 57) \bmod 101) \bmod 6 = 1$
$(((6 \times 70) + 38) \bmod 101) \bmod 6 = 0$

Neste exemplo, os valores de *a* e *b* foram sorteados para cada chave, e os mesmos devem ser armazenados caso se tenha interesse em buscar os dados inseridos na tabela. Esse fato é uma desvantagem da aplicação do espalhamento universal, pois caso os mesmos valores de *a* e *b* que foram utilizados para a inserção não sejam usados também na busca dos

dados, não será possível retornar o dado correto no momento de sua recuperação da tabela de espalhamento.

Cada posição da tabela possui uma lista encadeada que recebe os elementos associados a posição pelo método universal, resolvendo desta forma o problema das colisões. As listas das posições 2 e 4 estão vazias pois nenhum elemento foi associado as mesmas.

O método de espalhamento universal é bastante eficaz e estava sendo aplicado em versões iniciais do projeto prático deste trabalho, mas foi substituído pelo método de espalhamento perfeito com aplicação do método da divisão, pois o método não garante os resultados buscados por este trabalho.

6.6 A FUNÇÃO DE ESPALHAMENTO PERFEITO

Como mencionado na motivação do trabalho, um dos objetivos é encontrar a função de espalhamento perfeita, que consegue atribuir posições específicas para os dados na tabela de espalhamento sem a ocorrência de colisões. Quando uma função de espalhamento consegue atender este requisito, a função é perfeita, por isso recebe o nome de função de espalhamento perfeito.

Uma forma de se implementar a função de espalhamento perfeita é descrita a seguir:

- Uma tabela de espalhamento em dois níveis é construída. O primeiro nível se trata da tabela principal, e o segundo nível se trata de uma segunda tabela que deve armazenar os dados.

- Ambas as tabelas podem ser representadas com uma lista ou com vetores.

- Em cada um dos níveis é aplicado uma função de espalhamento para determinar a posição onde o dado será alocado na lista ou no vetor.

- A tabela do primeiro nível terá como tamanho a quantidade de dados que serão inseridos, e a tabela de segundo nível terá como tamanho o total de dados que se espalham naquele nível elevado ao quadrado. Por exemplo, se na primeira posição da tabela de primeiro nível for inserido 4 elementos, a tabela de segundo nível tera como tamanho $4^2 = 16$.

Os passos acima descrevem uma abordagem tradicional para se desenvolver o método de espalhamento perfeito, e é necessário ter cautela em sua utilização, pois uma quantidade de dados absurdamente grande pode ocasionar problemas de memória, podendo interromper a execução de um programa que utiliza este método.

Devido aos resultados obtidos com a aplicação da adaptação do método de transposição e o método da divisão para o segundo nível, não é necessário que as tabelas de segundo nível tenham como tamanho o quadrado do número de valores que se espalham na posição onde a tabela de segundo nível está, pois foi desenvolvida uma nova abordagem para realizar o espalhamento perfeito. A abordagem desenvolvida propõe utilizar o valor do nível máximo da malha + 1 como sendo o tamanho da tabela de primeiro nível, e para as tabelas de segundo nível, o tamanho será 4 elevado a posição onde a tabela de segundo nível se encontra na tabela de primeiro nível. Essa abordagem é possível pois a adaptação do método de transposição e o método da divisão conseguem alocar os dados de forma perfeita com as tabelas tendo os tamanhos descritos. E também não é necessário se preocupar com problemas de memória, pois neste trabalho é definido que uma malha terá como nível máximo o nível 12, sendo assim, a tabela de espalhamento armazenará 16.777.216 células, sendo um número de dados que a tabela de espalhamento consegue armazenar sem problema algum.

A Figura 10 mostra o esquema de uma tabela de espalhamento com o método de espalhamento perfeito em dois níveis, onde a tabela de primeiro nível armazena as tabelas de cada nível da malha, e as tabelas do segundo nível armazenam as células de cada nível da malha. Cada posição das tabela é numerada com o seu índice, partindo de 0.

Figura 10 - Tabela de espalhamento em dois níveis.

Na Figura 10, em vermelho está a tabela de primeiro nível, e em azul estão as tabelas do segundo nível. Para a tabela da posição 0 é alocado um único elemento pois $4^0 = 1$, para a tabela da posição 1 tem-se $4^1 = 4$, e para a tabela da posição 2 o tamanho alocado é $4^2 = 16$.

A respeito dos níveis da tabela e o nível da malha, são conceitos diferentes e não devem ser confundidos. Os níveis da tabela se referem as distribuições das listas que formam a tabela de espalhamento, como na Figura 10, onde em vermelho está a tabela de primeiro nível, e em azul estão as tabelas do segundo nível. E o nível da malha se refere a um valor numérico que determina a quantidade de células que podem ser alocadas na malha, por exemplo, uma malha uniforme de nível 2 terá $4^2 = 16$ células.

Tendo-se o esquema de agrupamento de tabelas definido, o Quadro 7 mostra a quantidade de células que cada nível de malha possui.

Quadro 7 - A quantidade de células por nível

Nível da Malha	**Total de Células**	**Total de Células ao Quadrado**
0	1	1
1	4	16
2	16	256
3	64	4096
4	256	65536
5	1024	1048576
6	4096	16777216
7	16384	268435456
8	65536	4294967296
9	262144	68719476736
10	1048576	$1{,}099511628 \times 10^{12}$
11	4194304	$1{,}759218604 \times 10^{13}$
12	16777216	$2{,}814749767 \times 10^{14}$

Estes valores foram obtidos seguindo o cálculo para se determinar a quantidade máxima de células em cada nível, que é 4^l onde l é o nível da malha atual. Ficam destacados os valores da coluna “Total de Células ao Quadrado”, pois a partir do nível 7 a tabela de espalhamento começou a ter problemas devido os valores estarem muito altos para se alocar espaço na tabela. Este foi outro motivo para não se aplicar o espalhamento universal, como é detalhado a seguir.

6.7 AS FUNÇÕES DE ESPALHAMENTO ESCOLHIDAS

Foram realizados testes de aplicação dos métodos de espalhamento na tabela, onde os métodos de transposição e divisão conseguiram garantir um bom espalhamento sem colisões e

com um tempo computacional O(1). Portanto, a aplicação destes dois métodos consegue satisfazer a necessidade de se ter o espalhamento perfeito.

O método de espalhamento universal apresenta um bom espalhamento de dados devido as escolhas aleatórias que são feitas para a realização do método, e é uma boa opção para ser aplicado não só em tabelas de espalhamento mas também em outras estruturas de dados. No entanto, a sua implementação teve que ser feita de uma forma diferente da abordagem que foi aplicada no espalhamento perfeito, pois para o espalhamento universal não apresentar colisões as tabelas no segundo nível devem ter seu tamanho elevado ao quadrado, e como o tamanho das tabelas é determinado por uma potência de 4, ao se elevar este valor ao quadrado, o tamanho total será muito grande conforme os níveis crescem de 7 em diante. E isso é ruim pois malhas uniformes que geram muitas células acabam causando uma interrupção na execução do programa devido ao espaçamento alocado que foi muito grande.

Outra questão sobre o espalhamento universal, é a necessidade de listas extras para armazenar os números primos que fazem parte do cálculo de índice, e os valores a e b, que também fazem parte do cálculo e são escolhido de forma aleatório, o que acaba elevando a complexidade da função devido a aplicação de métodos para gerar números pseudoaleatórios.

Devido a estes fatos, a função de espalhamento universal não foi aplicada na tabela de espalhamento, apenas a adaptação do método da transposição e o método da divisão. Lembrando que o método da transposição não é uma função de espalhamento como o método da divisão e a função de espalhamento universal, é apenas um método para se calcular a chave intermediária que é aplicada em outros métodos, mas apesar disso, pode ter seu resultado aplicado diretamente como uma posição para um dado na tabela de espalhamento. Após as escolhas das funções de espalhamento serem definidas, foram realizados testes com inserção de elementos até o nível 12, e os dados foram espalhados sem a ocorrência de colisões.

6.8 A IMPLEMENTAÇÃO DAS FUNÇÕES ESCOLHIDAS

Ainda neste tópico, já foram apresentados os métodos numéricos de espalhamento, agora será detalhada a implementação destes métodos com seus respectivos pseudocódigos e as complexidades de cada função.

6.8.1 O Cálculo da Chave Intermediária

No Quadro 8 segue o pseudocódigo para o cálculo da chave intermediária com a complexidade de cada instrução:

Quadro 8 - Pseudocódigo para o cálculo da chave intermediária

	Código	Complexidade
1	**função** calculoChave(índiceX, índiceY, nívelL)	
2	colunasNível = 2^nívelL	O(1)
3	chave = (colunasNível * índiceY) + índiceX	O(1)
4	**retorna** chave	
5	**fim função**	

Na implementação do projeto, a operação de exponenciação (representada no pseudocódigo pelo símbolo ^ na linha 2) é calculada pela função pow. A complexidade total deste método é O(1).

6.8.2 O Método da Divisão

No Quadro 9 segue o pseudocódigo do método da divisão e a complexidade de cada instrução:

Quadro 9 - Pseudocódigo do método da divisão

Código	Função
h = calculoChave(índiceX, índiceY, nívelL) % tamanhoTabela	O(1)

Como definido anteriormente, o método *calculoChave* tem tempo O(1), e a operação de resto da divisão também é definida com tempo O(1), portanto, o método da divisão também tem tempo O(1). A variável *tamanhoTabela* representa o tamanho total da tabela de segundo nível, que foi determinado previamente e nunca é alterado. Então, a quantidade de elementos em uma tabela não altera a complexidade total.

7 A IMPLEMENTAÇÃO DA TABELA DE ESPALHAMENTO

Após as devidas definições, foi feita a implementação da tabela de espalhamento. Para representar a tabela e suas operações, foi feita uma classe que contém os métodos de espalhamento selecionados, as operações de inserção, remoção e busca, e as estruturas que armazenam os dados.

7.1 O ARMAZENAMENTO DOS DADOS

Os dados são armazenados em um vetor, porém, como está sendo aplicado o espalhamento perfeito, é necessário representar a tabela em dois níveis. Isso é feito utilizando um vetor de vetores. Dessa forma, o vetor do primeiro nível armazena os vetores do segundo nível. Os vetores do segundo nível armazenam os dados que são espalhados.

Como descrito na introdução do trabalho, os vetores são representadas com a biblioteca *vector* do C++, sendo assim, os níveis da tabela são representados como um *vector* de *vectors* (vetor de vetores) na implementação.

Quadro 10 - Um vetor de vetores com a estrutura *vector*

```
std::vector<std::vector<Celula>> tabela;
```

A estrutura no Quadro 10 terá como tamanho no primeiro nível, o nível total da malha, e no segundo nível, o tamanho será 4 elevado ao índice atual da malha que vai de 0 ao nível total da malha. E este tamanho deve ser previamente inicializado, e isso é feito com um laço de repetição que varia de $i = 0$ até o nível total da malha. Então, a tabela da posição i recebe como tamanho 4^i . Este processo é necessário devido a requisição do tamanho total das tabelas de segundo nível para a aplicação das funções de espalhamento.

7.2 A IMPLEMENTAÇÃO DAS OPERAÇÕES

7.2.1 Definições

Todas as operações descritas nesta seção terão a representação em pseudocódigo para se ter um embasamento de como cada operação funciona. Lembrando que a tabela de espalhamento possui dois níveis, e para acessar o primeiro nível é utilizado o nível da célula que é representado pela variável *índiceL*.

A variável *índiceH* é utilizada para armazenar o valor do índice da célula na tabela de segundo nível, e está utilizando o método da divisão para calcular o índice com a chave intermediária, assim, esta variável está presente nos pseudocódigos de todas as operações aqui descritas.

O método de cálculo de chave intermediária já foi descrito e demonstrado que sua complexidade é O(1).

7.2.2 Inserção

A operação de inserção fica responsável por inserir um elemento na tabela, verificando a ocorrência de uma colisão. A inserção é feita recebendo uma célula e utilizando seus índices para se calcular o índice da tabela de espalhamento. Após o cálculo de índice ser efetuado, é verificado se a posição na tabela de segundo nível contém uma célula nula ou não. Esta verificação é feita analisando se o parâmetro *nulo* é verdadeiro ou falso. Se *nulo* for verdadeiro, não houve colisão pois a posição da tabela contém uma célula nula, e a nova célula é inserida em seu lugar. Caso contrário, houve uma colisão e uma mensagem de erro é reportada interrompendo a execução do programa.

No Quadro 11 segue a representação do método de inserção em pseudocódigo, com as complexidades de cada operação:

Quadro 11 - Pseudocódigo do método de inserção

1	**função** inserir(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**se** !tabela[índiceL][índiceH].nulo **então**	O(1)
4	Houve uma colisão e interrompe a execução	O(1)
5	**fim se**	
6	tabela[índiceL][índiceH] = {índiceX, índiceY, índiceL}	O(1)
7	**fim função**	

Como todas as operações da função tem complexidade constante, a operação de inserção é realizada em tempo O(1).

A verificação realizada na linha 3 é feita por segurança, caso uma célula que já está presente na tabela de espalhamento seja inserida novamente por algum descuido, será detectada uma colisão. Caso ocorra uma colisão, a instrução da linha 6 não será executada pois o programa é interrompido. Também vale lembrar que para a tabela de primeiro nível não

é aplicado nenhum método de espalhamento, pois o espalhamento para o primeiro nível é feito de forma direta utilizando o índice *l*.

7.2.3 Remoção

A operação de remoção realiza a substituição de uma célula em uma posição da tabela por uma célula nula. Essa operação é feita dessa forma pois não deve ser reduzido o tamanho total da tabela, e nenhuma posição deve deixar de existir, pois outro elemento pode ser inserido em seu lugar.

Para realizar a remoção, também é calculado o índice da célula na tabela de espalhamento, que utiliza os parâmetros que estão na célula que foi passada ao método de remoção. Após o índice ser calculado, a célula na posição do índice na tabela de segundo nível é substituída pela célula nula. A célula nula é definida com seus três índices em 0 e o parâmetro *nulo* em verdadeiro.

No Quadro 12 segue a representação do método de remoção em pseudocódigo com sua complexidade:

Quadro 12 - Pseudocódigo do método de remoção

1	**função** remover(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	tabela[índiceL][índiceH] = {0, 0, 0, verdadeiro}	O(1)
4	**fim função**	

7.2.4 Busca

A operação de busca retorna uma célula da tabela de espalhamento, de acordo com os índices passados ao método. Então, basta calcular o índice da célula e retorná-la ao fim do método. Caso não exista nenhuma célula na posição, é retornada uma célula nula.

No Quadro 13 segue a representação do método de busca em pseudocódigo com sua complexidade:

Quadro 13 - Pseudocódigo do método de busca

1	**função** buscaCelula(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**retorna** tabela[índiceL][índiceH]	O(1)

4	**fim função**	

7.2.5 Contém Célula

A operação contém célula deve indicar se a tabela contém uma célula ou não. Para o método é passado os 3 valores de indexação da célula, e então é calculado índice da célula na tabela. Ao ter o índice definido, basta comparar se os valores x, y e l da célula na tabela são iguais aos valores x, y e l passados ao método. Se sim, o método retorna verdadeiro, senão retorna falso. Esta operação é empregada para verificar se uma célula que foi inserida na tabela está de fato na tabela.

No Quadro 14 segue a representação do método contém célula em pseudocódigo com sua complexidade:

Quadro 14 - Pseudocódigo do método contém célula

1	**função** contemCelula(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**retorna** ((tabela[índiceL][índiceH].x == x) e (tabela[índiceL][índiceH].y == y) e (tabela[índiceL][índiceH].l == l))	O(1)
4	**fim função**	

8 EXPERIMENTOS E RESULTADOS

8.1 EXPERIMENTO - A FORMAÇÃO DE CÉLULAS

O uso real das malhas terão as células fornecidas pela aplicação em que a tabela de espalhamento será aplicada, portanto, para se testar as implementações construídas, era necessário produzir as células que seriam inseridas na tabela. E isso foi feito com implementação de uma estratégia para gerar as células necessárias para os testes. A partir dessa necessidade, foi feito este experimento que utiliza um algoritmo que espalha as células até um determinado nível.

Como descrito em seções anteriores, uma célula é definida por $c = (x, y, l)$, onde x, y e l indexam a célula, sendo l o nível ao qual a célula pertence. O nível da célula é definido previamente. Quando se deseja criar uma malha de nível único, o valor do nível é constante. Quando a malha deve armazenar células de mais de um nível, o nível deve variar em um intervalo que inicia no nível mínimo até o nível máximo que se deseja representar. O intervalo é feito com o percurso de um laço de repetição do nível mínimo até o nível máximo. Ambos os valores de nível devem ser positivos, e o nível mínimo pode iniciar em 0.

Para definir os índices x e y, também é feito um percurso através de um laço de repetição, iniciando em 0 até 4^l para garantir o número total de células em cada nível. Dentro do laço de repetição, o índice y é incrementado, e quando o valor de y for igual a $\sqrt{4^l}$ o índice x é incrementado e o índice y é zerado.

Os índices são incrementados desta forma pois é necessário garantir que não serão gerados índices que não correspondem ao nível atual.

Como um pequeno exemplo, dada uma malha uniforme de nível 2, o número de células geradas será $4^2 = 16$ que também é o tamanho da tabela de segundo nível da posição 2. Porém os índices x e y não serão incrementados até 16, pois quando o valor de y for igual a $\sqrt{4^2} = 4$ o índice x é incrementado e o índice y é zerado, portanto, os índices x e y crescem no intervalo [0,3] no nível 2.

No Quadro 15 segue um pseudocódigo que apresenta as configurações descritas acima:

Quadro 15 - Pseudocódigo para o experimento de formação de células

1 **2**	indiceX = 0 indiceY = 0

```
3  nivelAtual = 2
4  para k de 0 ate 4 ^ nivelAtual faça
5     célula = {indiceX, indiceY, nivelAtual}
6     indiceY++
7     se indiceY == ((4 ^ nivelAtual) ^ (½)) então
8       indiceY = 0
9       indiceX++
10    fim se
11 fim para
```

8.2 EXPERIMENTO - GERAÇÃO ALEATÓRIA DE CÉLULAS

Após a construção da tabela de espalhamento, foi realizado um experimento para se gerar 3.000.000 (três milhões) de células de forma totalmente aleatória. Para isso, foi utilizada a função rand da biblioteca <cstdlib> para se gerar números aleatórios dentro de laços de repetição que garantissem que o número total de células seria gerado.

No Quadro 16 segue o pseudocódigo da implementação deste experimento:

Quadro 16 - Pseudocódigo do experimento de geração aleatória de células

```
1  numeroDeCelulas = 3000000
2  nivelMaximo = 11
3  tabela = Tabela(nivelMaximo)
4  enquanto tabela.getTotalElementos() < numeroDeCelulas faça
5     para i de 0 até numeroDeCelulas faça
6        se tabela.totalDeElementos() == numeroDeCelulas então
7           fim para
8        fim se
9        índiceL = rand() % nivelMaximo+1
10       fator = pow(2, l)
11       índiceX = rand() % fator
12       índiceY = rand() % fator
13       se tabela.contemCelula({índiceX, índiceY, índiceL}) == falso então
14          tabela.inserir({índiceX, índiceY, índiceL})
15       fim se
16    fim para
17 fim enquanto
```

Como o número de células a ser gerado é muito grande, foi necessário preparar a tabela para alocar células até o nível 11.

Embora este experimento seja simples, foi importante para poder se visualizar o comportamento da geração de células em meio a imposição de uma geração randômica. E devido ao fato do número de células geradas ser bastante alto, a malha para armazenar as células seria não-uniforme e de tamanho 4^{11} = 4194304.

É importante destacar que o método *contemCelula* na linha 13 está verificando se a célula já foi inserida na tabela, e essa verificação é necessária pois se for passada uma célula que já está na tabela para o método *inserir*, vai ser reportada uma colisão e o experimento falha. E esta verificação também é necessária pois como as células estão sendo geradas de forma aleatória, uma mesma célula pode ser gerada diversas vezes.

Outra observação importante refere-se a função *rand()*, que se trata da função para a geração de número aleatórios.

8.3 REVISÃO DO FUNCIONAMENTO DA ESTRATÉGIA DE ESPALHAMENTO

Durante o processo de implementação da tabela de espalhamento, foram feitas muitas análises a respeito dos métodos de espalhamento que seriam aplicados. A princípio, o método de espalhamento universal apresentava o resultado mais aceitável para se aplicar na tabela. Porém, este método tinha a desvantagem de se necessitar realizar um espalhamento prévio apenas para se ter conhecimento de onde os elementos seriam alocados, e não era possível espalhar os dados sem realizar este processo, pois a função de espalhamento universal não fornecia nenhuma estratégia de alocação sem o conhecimento prévio dos dados.

É importante lembrar que o espalhamento dos dados ocorre de forma muito melhor se os mesmos forem previamente conhecidos, pois é possível aplicar estratégias como o espalhamento prévio para se determinar as posições finais dos dados, no entanto, após muitos testes e verificações, foi encontrada a abordagem perfeita de espalhamento.

Com o uso do próprio nível da célula para realizar o endereçamento direto no primeiro nível, foi necessário aplicar uma função de espalhamento apenas no segundo nível. As funções escolhidas foram uma adaptação do método de transposição de células aplicado de forma direta, e o método da divisão. Ambos os métodos apresentaram resultados satisfatórios, portanto, os dois são aptos para realizar o espalhamento, mas vale lembrar que a adaptação do método de transposição realiza o cálculo da chave intermediária.

Tendo sido escolhidas as funções para aplicar na tabela de espalhamento, foram analisados os resultados de inserção de células, e foi observado que a abordagem aplicada consegue manter as células da malha ordenadas na tabela, e isso é muito interessante pois os dados podem ser organizados de forma padrão dentro da tabela, e essa organização é definida em ordem crescente dos índices da célula, como mostra a Figura 11:

Malha de nível 1:
[1][0] = {0,0,1}
[1][1] = {1,0,1}
[1][2] = {0,1,1}
[1][3] = {1,1,1}

Figura 11 - Um exemplo de como os dados são dispostos com o espalhamento perfeito.

Para uma malha de nível 1, as 4 células possíveis da malha estão espalhadas de forma crescente, sendo as células ordenadas primeiramente de acordo com o índice x, e logo após pelo índice y. E esse ordenamento ocorre nas tabelas de segundo nível, pois as tabelas de primeiro nível são ordenadas pelos níveis das células, ou seja, a célula do nível 0 é a primeira célula da tabela, logo após são inseridas as células do nível 1, do nível 2, e assim por diante.

Após concluir todas as análises, observar os resultados e tendo sida construída uma tabela de espalhamento com operações que respeitam o tempo de complexidade constante, é enfim concluída a pesquisa por uma função de espalhamento com baixa complexidade e sem a ocorrência de colisões, ou seja, uma função de espalhamento perfeita. Mas não foi encontrado apenas uma função, e sim toda uma metodologia que soluciona a questão principal deste trabalho.

9 AS VIZINHANÇAS

Agora será tratada a questão das vizinhanças em uma malha, que se refere ao processo de busca pelas células vizinhas de uma célula atual *c*.

Ao ser feita uma busca pelos vizinhos de uma determinada célula, as células vizinhas que são encontradas são armazenadas em uma lista no método de busca pelos vizinhos, que pertence a classe que representa a malha. Inicialmente, o armazenamento das células vizinhas seria feito em um hipergrafo, no entanto, durante o processo de construção do algoritmo de pesquisa das células vizinhas, foi constatado que o hipergrafo seria uma estrutura muito complexa para armazenar as células encontradas. E como a operação de busca pelas células vizinhas já tem uma complexidade mais elevada, apenas uma lista para armazenar as células encontradas já foi o suficiente.

9.1 A PRIMEIRA IDEIA PARA O CÁLCULO DAS VIZINHANÇAS

O hipergrafo se trata de um grafo no qual as arestas podem conectar qualquer número de vértices, sendo cada aresta uma hiperaresta. Este conceito é importante pois uma célula pode conter uma grande quantidade de vizinhos. Uma ideia de implementação que justificaria o uso do hipergrafo, é a construção prévia do hipergrafo com todas as vizinhanças já determinadas, assim, o hipergrafo seria um mapa que conteria os índices das células na tabela de espalhamento como indexador, e um conjunto contendo os índices das células vizinhas na tabela de espalhamento. Uma tupla com os índices de uma célula seria o vértice, e as arestas seriam formadas entre o vértice que indexa um conjunto no mapa, e os vértices contidos neste mesmo conjunto.

(0,1,1) [1,2]	(1,1,1) [1,3]
(0,0,1) [1,0]	(1,0,1) [1,1]

```
Vertice_Celula {
    indicePrimeiroNivel;
    indiceSegundoNivel;
};

Hipergrafo[Vertice_Celula{1,0}] = {
    Vertice_Celula{1,1},
    Vertice_Celula{1,2},
    Vertice_Celula{1,3},
};
```

Figura 12 - Uma possível aplicação de hipergrafo para armazenar as células vizinhas.

Na Figura 12 segue um exemplo de como poderia ser implementado um hipergrafo, onde cada índice do mapa possui um conjunto de vértices, todos conectados entre si, o que forma um subgrafo que se torna uma aresta do hipergrafo. Abaixo dos índices das células, estão destacados nos colchetes os índices das células na tabela de espalhamento, onde para $[i,j]$, i é o índice da célula na tabela de primeiro nível, e j é o índice da célula na tabela de segundo nível. Os vértices do hipergrafo não são formados pelas células em si, pois não faria sentido ter uma segunda estrutura para armazenar as células durante a execução do programa, sendo que a tabela de espalhamento já assume este papel.

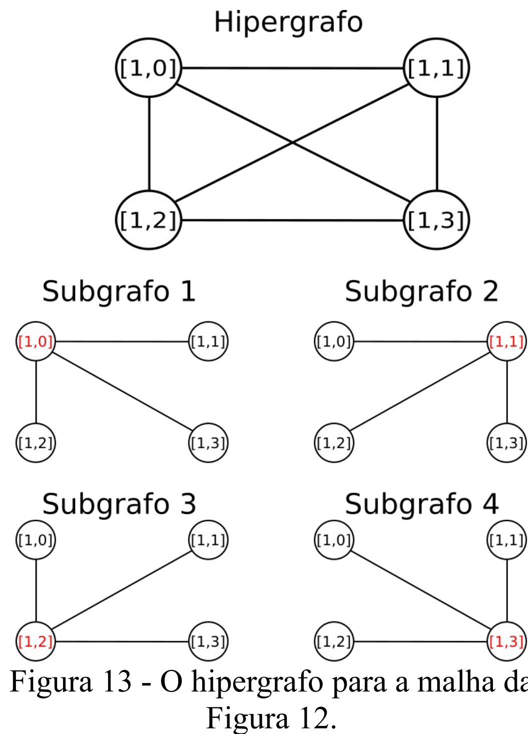


Figura 13 - O hipergrafo para a malha da Figura 12.

Na Figura 13 é apresentado o hipergrafo para a malha da figura 12, onde cada vértice corresponde a tupla de índices da célula na tabela de espalhamento, e cada aresta conecta os vértices que representam as células que possuem adjacência. Abaixo do hipergrafo, estão os subgrafos que são gerados para as quatro células da malha, e em vermelho está destacado o vértice que será buscado para retornar seus vizinhos, portanto, seguindo a implementação de hipergrafo por mapas proposta na figura 12, o vértice em vermelho se trada vértice que indexa um subgrafo no mapa.

Após o hipergrafo ser construído, é passada uma célula c que se deseja encontrar os vizinhos, e então são calculados os índices de c na tabela de espalhamento, e então com os índices calculados, é feita a consulta no hipergrafo para se retornar o conjunto de vizinhos da célula c. Ao ser obtido o conjunto de vizinhos, as células podem ser obtidas da tabela de espalhamento passando os índices contidos em cada vértice no conjunto, através do método

operator[i][j], que é implementado pela classe *vector* da STL do C++. Por exemplo, se for feita uma busca pelos vizinhos do vértice [1,0], será retornado o conjunto {[1,1], [1,2], [1,3]}, e então com cada índice contido no conjunto, pode ser obtida a célula da tabela de espalhamento.

Embora a ideia apresentada seja muito prática para o armazenamento das vizinhanças, a mesma foi descartada, pois para se armazenar as vizinhanças de todas as células no hipergrafo, seria necessário percorrer toda a tabela de espalhamento, o que seria extremamente ineficiente. Portanto, a procura de vizinhos de uma célula na malha ocorre toda vez que a operação de busca de vizinhos for requisitada, pois dessa forma o algoritmo não necessita percorrer toda a tabela de espalhamento.

Nos próximos tópicos seguem os detalhes do algoritmo e da implementação da busca de células vizinhas.

9.2 OPERAÇÕES NECESSÁRIAS PARA A BUSCA DE CÉLULAS VIZINHAS

Antes de entrar na estratégia de busca de células vizinhas, é importante destacar duas operações essenciais para este processo, que são as operações de expansão e retração de uma célula.

9.2.1 A Expansão de uma Célula

A operação de expansão de uma célula é o processo de se expandir uma célula de um nível l em quatro novas células do próximo nível ($l+1$). Este processo é feito a partir dos parâmetros (x,y,l) existentes na célula. Portanto, para uma célula $c = (x,y,l)$, o algoritmo realiza os seguintes passos:

1. São obtidos novos parâmetros $x1$, $y1$ e $l1$, onde:
   - $x1 = 2 \times x$
   - $y1 = 2 \times y$
   - $l1 = l + 1$
2. São formadas as quatro novas células $\{c1, c2, c3, c4\}$ a partir de c, que terão os seguintes valores:
   - $c1 = (x1, y1, l1)$
   - $c2 = (x1, y1+1, l1)$
   - $c3 = (x1+1, y1, l1)$

- *c4 = (x1+1, y1+1, l1)*

3. Após a formação das novas células, as mesmas são retornadas pelo método em uma lista.

Em um pequeno exemplo, tem-se interesse em expandir a célula *c = (0,1,1)*. A partir de *c*, são formadas as células {*c1*, *c2*, *c3*, *c4*}:

1. São obtidos os novos parâmetros:
   - $x1 = 2 \times 0 = 0$
   - $y1 = 2 \times 1 = 2$
   - $l1 = 1 + 1 = 2$
2. São formadas as quatro novas células:
   - *c1 = (0, 2, 2)*
   - *c2 = (0, 2+1, 2) = (0, 3, 2)*
   - *c3 = (0+1, 2, 2) = (1, 2, 2)*
   - *c4 = (0+1, 2+1, 2) = (1, 3, 2)*
3. Ao final é retornada a lista com as novas células.

Quadro 17 - O pseudocódigo para a expansão de uma célula

```
1  função expandir(célula)
2    x = 2 * célula.x
3    y = 2 * célula.y
4    l = célula.l + 1
5    Célula c1 = {x, y, l}
6    Célula c2 = {x, y + 1, l}
7    Célula c3 = {x + 1, y, l}
8    Célula c4 = {x + 1, y + 1, l}
9    retorna Lista{c1, c2, c3, c4}
10 fim função
```

9.2.2 A Retração de uma Célula

A operação de retração de uma célula segue o processo contrário à operação de expansão. A partir de uma célula *c = (x,y,l)*, tem-se interesse em descobrir uma célula *c1 = (x1,y1,l1)* que ocupa a região de *c* no nível anterior. Para isso, são realizados os seguintes passos:

1. Se *x* for par:

- $x1 = x \div 2$

2. Se *x* for impar:
   - $x1 = (x - 1) \div 2$
3. Se *y* for par:
   - $y1 = y \div 2$
4. Se *y* for impar:
   - $y1 = (y - 1) \div 2$
5. O nível *l1* é o decremento de *l*:
   - $l1 = l - 1$
6. Ao executar todos os passos, a célula *c1* é retornada pelo método.

Para um exemplo, será aplicado o algoritmo de retração sob a célula *c = (0,3,2)*:

1. 0 é par:
   - $x1 = 0 \div 2 = 0$
2. 3 é ímpar:
   - $y1 = (3-1) \div 2 = 1$
3. O novo nível é obtido:
   - $l1 = 2 - 1 = 1$
4. É retornada a célula *c1 = (0,1,1)*.

Quadro 18 - O pseudocódigo para a retração de uma célula

1	**função** retrair(célula)
2	x = ((célula.x % 2 == 0) ? célula.x / 2 : (célula.x - 1) / 2)
3	y = ((célula.y % 2 == 0) ? célula.y / 2 : (célula.y - 1) / 2)
4	l = célula.l - 1
5	**retorna** Célula{x, y, l}
6	**fim função**

Nas linhas 2 e 3 foram utilizados operadores ternários para decidir qual cálculo deve ser feito de acordo com a paridade do índice.

9.2.3 Observações

Estes algoritmos são necessários para as operações de busca de células vizinhas, pois caso não exista alguma célula na tabela de espalhamento que deveria ser a célula vizinha, esta

célula que não foi encontrada será expandida e retraída até se encontrar as células que colidem com a célula de interesse para se determinar as vizinhanças.

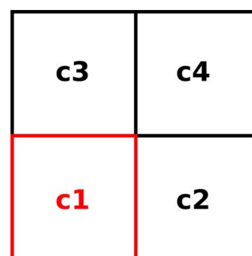
Estes algoritmos não geram alterações na malha e na tabela de espalhamento, apenas realizam suas devidas funções e retornam as células desejadas. Porém, outros métodos que serão apresentados nos próximos capítulos fazem uso destes algoritmos para realizar modificações na malha para determinado interesse. Então, estes algoritmos assumem papeis auxiliares importantes para operações em malha.

9.3 A ESTRATÉGIA DE BUSCA DE CÉLULAS VIZINHAS

Dada uma célula $c = (x,y,l)$, para se encontrar as células vizinhas de *c*, deve ser feita uma busca em dois passos na malha. O primeiro passo consiste em avaliar as células vizinhas da célula *c* e que estão no mesmo nível da malha. Caso não exista alguma ou nenhuma célula que colida com *c* no mesmo nível, o algoritmo realiza o segundo passo de busca onde é feita uma busca recursiva até ser encontrada uma célula que colida com *c* (esta célula pode estar em um nível de malha anterior ou próximo ao nível de *c*).

9.3.1 O Primeiro Passo de Busca

A busca no primeiro passo é feita de forma simples, pois basta aplicar operações aritméticas básicas nos índices de *c* para formar as células que colidem com *c* no mesmo nível de malha, e caso estas células estejam incluídas na tabela de espalhamento significa que elas existem naquele nível de malha, portanto são adicionadas na lista de células vizinhas de *c*. A Figura 14 ilustra as operações realizadas para se encontrar as células vizinhas no mesmo nível de malha.



$c1 = (x, y, l)$
$c2 = (x+1, y, l)$
$c3 = (x, y+1, l)$
$c4 = (x+1, y+1, l)$

Figura 14 - Aplicação da busca de células vizinhas em uma malha uniforme de nível 1.

Na Figura 14, a célula *c1* é destacada em vermelho pois se trata da célula que se deseja buscar os vizinhos. As células *c2*, *c3* e *c4* são as células vizinhas pois estas colidem com a célula *c1*. Para a busca ser realizada, as células *c2*, *c3* e *c4* são formadas a partir dos índices *(x,y,l)* pertencentes a célula *c1*, e estes índices são também coloridos em vermelho para indicar sua origem.

A regra de formação das células é muito clara, como *c2* é uma célula que está a direita de *c1*, basta somar a coordena x de *c1* com 1 para se obter a coordenada x de *c2*. Já para *c4*, são somadas as duas coordenadas pois *c4* está na diagonal superior direita. Caso existisse uma célula a esquerda de *c1*, a coordenada x desta célula seria formada subtraindo a coordenada x de *c1* por 1. Como a malha do exemplo é de nível 1, não foi necessário realizar operações de subtração, mas para malhas de níveis maiores onde existem mais células, são necessárias as aplicações das operações de adição e subtração para se formar todas as células vizinhas. A Figura 15 abaixo apresenta um exemplo de busca de uma célula em uma malha de nível 2.

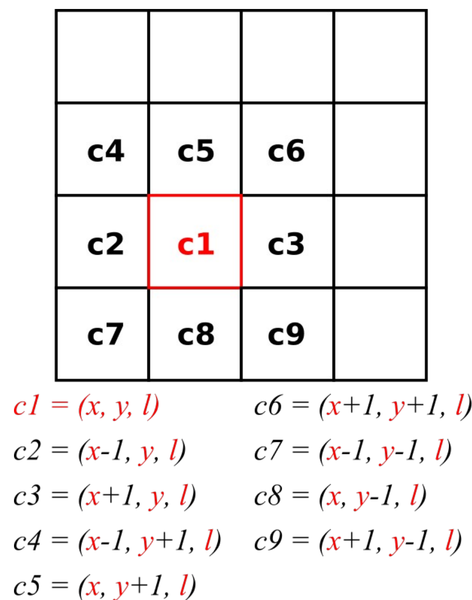


Figura 15 - A busca de células vizinhas para uma malha uniforme de nível 2.

Na Figura 15, a célula *c1* é a célula de interesse para se encontrar as células vizinhas, e como *c1* se encontra em uma região central da malha de nível 2, são necessárias mais operações para se encontrar todas as células que colidem com *c1*.

Este primeiro passo de busca é suficiente para se encontrar todas as células vizinhas em uma malha uniforme, mas caso alguma das células formadas não estivesse na tabela de espalhamento, significaria que a célula não está presente na malha, portanto, a malha seria não-uniforme, e isso leva ao segundo passo de busca que trata da busca por células em uma vizinhança de malha não-uniforme.

9.3.2 O Segundo Passo de Busca

O segundo passo de busca é ativado quando deve-se procurar as células vizinhas em uma malha não-uniforme. Caso alguma célula formada no primeiro passo de busca não exista na tabela de espalhamento, e por consequência, na malha, esta célula que não existe é expandida para as células do próximo nível, e então é verificado se as células que colidem com a célula de busca atual existem na tabela de espalhamento, e caso existam, estas células são adicionadas a lista de células vizinhas, caso não existam, o processo é repetido recursivamente passando as células que não existem para ser feita a procura no próximo nível.

O processo de busca com a expansão das células que não foram encontradas na tabela de espalhamento é feito para procurar os vizinhos no próximo nível. Mas, também deve ser feita a procura nas células do nível anterior, pois também podem existir células vizinhas no nível anterior. Então, a célula que não foi encontrada no mesmo nível que a célula de interesse da busca, deve ser retraída para a célula de nível anterior, onde será feita uma analise de colisão com as células daquele nível, para se encontrar as células vizinhas. Este processo também é recursivo.

Na subseção 9.3.4 são descritos os algoritmos para a busca das células vizinhas.

9.3.3 Teste de Mesa

A partir da Figura 16 será feito um teste de mesa onde se tem interesse em descobrir quais são as células vizinhas da célula (4,3,3):

Figura 16 - Uma malha não-uniforme de nível 3 onde a célula (4,3,3) é a célula de interesse.

Para se encontrar as células vizinhas de (4,3,3), o processo é iniciado com o primeiro passo de busca tendo os seguintes resultados:

Quadro 19 - Resultados do primeiro passo de busca para a malha da Figura 16

c1 = (4, 3, 3)	c6 = (4 + 1, 3 + 1, 3) = (5, 4, 3)
c2 = (4 - 1, 3, 3) = (3, 3, 3)	c7 = (4 - 1, 3 - 1, 3) = (3, 2, 3)
c3 = (4 + 1, 3, 3) = (5, 3, 3)	c8 = (4, 3 - 1, 3) = (4, 2, 3)
c4 = (4 - 1, 3 + 1, 3) = (3, 4, 3)	c9 = (4 + 1, 3 - 1, 3) = (5, 2, 3)
c5 = (4, 3 + 1, 3) = (4, 4, 3)	

Como a célula (4,3,3) está no nível mais fino da malha (ou seja, o nível máximo) não se tem a necessidade de buscar vizinhos no próximo nível, pois este não existe. Então, o algoritmo realiza a busca no nível anterior ao nível da célula (4,3,3), que é o nível 2. Para isso, o algoritmo de busca de vizinhos invoca o algoritmo de retração em cada célula que não foi encontrada no primeiro passo de busca para se descobrir qual a célula que está na região da mesma no nível anterior. Neste teste de mesa, as células encontradas no primeiro passo foram coloridas em azul, e as células não encontradas estão em preto na tabela.

Portanto, para as células (3,3,3), (3,4,3), (4,4,3) e (3,2,3), o algoritmo retrairá as células, até encontrar uma célula que exista na tabela de espalhamento, sendo esta célula uma vizinha de (4,3,3). Para as células (3,3,3) e (3,2,3), o algoritmo de retração formou a célula (1,1,2), que existe na malha e colide com a célula (4,3,3). Para a célula (3,4,3), o algoritmo formou a célula (1,2,2), que não existe na malha, então, a célula (1,2,2) é retraída formando a célula (0,1,1), que está na malha e é vizinha de (4,3,3). Por fim, para a célula (4,4,3), o algoritmo de retração formou a célula (2,2,2), que não existe na malha, então é novamente retraída, formando a célula (1,1,1), que está presente na malha e é vizinha de (4,3,3).

O algoritmo em dois passos conseguiu encontrar todas as células vizinhas de (4,3,3) em uma malha não-uniforme, como mostra a Figura 17:

(0,1,1) (1,1,1)

(0,1,2) (1,1,2) (4,3,3) (5,3,3) (4,2,3) (5,2,3) (3,1,2)

(0,0,2) (1,0,2) (2,0,2) (3,0,2)

Figura 17 - As células vizinhas de (4,3,3).

Estão coloridas em azul todas as vizinhas de (4,3,3).

O outro caso interessante de busca de vizinhos de uma célula, é quando as células vizinhas estão em um nível superior a célula de interesse. Por exemplo, para a malha da Figura 17, a célula (1,1,2) teria como vizinha a direita a célula (2,1,2), porém, esta não existe na malha. Então, o segundo passo do algoritmo faria a retração e a expansão da célula (2,1,2) para encontrar as células vizinhas nos níveis anterior e posterior. Como nenhuma célula será encontrada no nível anterior para (2,1,2) o algoritmo continua apenas com as células do nível posterior. E aplicando-se a expansão da célula (2,1,2), são formadas as células (4,2,3), (4,3,3), (5,2,3) e (5,3,3). Todas estas células existem na malha, mas apenas duas delas são vizinhas de (1,1,2), e para saber quem são as vizinhas, o algoritmo escolhe as duas células que estão a direita de (1,1,2), pois (2,1,2) também estaria a direita caso existisse na malha.

9.3.4 O Pseudocódigo Para as Estratégias de Busca de Células Vizinhas

Neste tópico são apresentados os pseudocódigos dos métodos que realizam a busca de células vizinhas.

Quadro 20 - A função principal para a busca de células vizinhas

```
1  função vizinhos(índiceX, índiceY, índiceL)
2     vizinhos = lista de células
3     fator = ((4 ^ índiceL) ^ (1 / 2)) - 1
4
5     se  índiceY > 0 então
6        Célula c = {índiceX, índiceY - 1, índiceL}
7        inserirVizinho(c, vizinhos, 1)
```

```
8    fim se
9
10   se índiceX > 0 então
11      Célula c = {índiceX - 1, índiceY, índiceL}
12      inserirVizinho(c, vizinhos, 2)
13   fim se
14
15   se índiceX < fator então
16      Célula c = {índiceX + 1, índiceY, índiceL}
17      inserirVizinho(c, vizinhos, 3)
18   fim se
19
20   se índiceY < fator então
21      Célula c = {índiceX, índiceY + 1, índiceL}
22      inserirVizinho(c, vizinhos, 4)
23   fim se
24
25   se índiceX > 0 e índiceY > 0 então
26      Célula c = {índiceX - 1, índiceY - 1, índiceL}
27      inserirVizinho(c, vizinhos, 5)
28   fim se
29
30   se índiceX > 0 e índiceY < fator então
31      Célula c = {índiceX - 1, índiceY + 1, índiceL}
32      inserirVizinho(c, vizinhos, 6)
33   fim se
34
35   se índiceX < fator e índiceY> 0 então
36      Célula c = {índiceX + 1, índiceY - 1, índiceL}
37      inserirVizinho(c, vizinhos, 7);
38   fim se
39
40   se índiceX < fator e índiceY < fator então
41      Célula c = {índiceX + 1, índiceY + 1, índiceL}
42      inserirVizinho(c, vizinhos, 8)
43   fim se
44
45   retorna vizinhos
46 fim função
```

Na função *vizinhos* é inciado o primeiro passo de busca por células vizinhas da célula de interesse. As células vizinhas são calculadas através de operações básicas e enviadas a função *inserirVizinho*.

Cada condição determina a direção em que células vizinhas deverão ser buscadas na função *inserirVizinho*, porém essa busca será realizada apena se a célula *c* não existir na tabela

de espalhamento, e a direção só interessa para a busca de células em níveis superiores, não sendo necessária para a busca em níveis anteriores ao nível da célula de interesse.

Quadro 21 - O método *inserirVizinho*

```
1  função inserirVizinho(célula, vizinhos, direção)
2     se tabela.contemCelula(célula) então
3        vizinhos.inserir(célula)
4     senão
5        procuraVizinhosProximoNivel(célula, vizinhos, direção)
6        procuraVizinhosNivelAnteior(célula, vizinhos)
7     fim se
8  fim função
```

A função *inserirVizinho* deve inserir a célula obtida no primeiro passo de busca na lista de vizinhos, caso esta célula exista na tabela de espalhamento. Se a célula não existir na tabela, a função inicia o segundo passo de busca chamando as funções *procuraVizinhosProximoNivel* e *procuraVizinhosNivelAnteior*. Isso acontece pois se a célula não existe na tabela de espalhamento, significa que podem existir células nos níveis anteriores e sucessores que ocupem sua região, e estas células vem a ser as células vizinhas.

Quadro 22 - O método *procuraVizinhosProximoNivel*

```
1  função procuraVizinhosProximoNivel(célula, vizinhos, direção)
2     se célula.l < nivelMalha então
3        células = vizinhosProximoNivel(célula, direção)
4        para c em células faça
5           se tabela.contemCelula(c) e !c.nulo então
6              vizinhos.inserir(c);
7           senão
8              se c.l < nivelMalha e !c.nulo então
9                 procuraVizinhosProximoNivel(c, vizinhos, direção)
10             fim se
11          fim se
12       fim para
13    fim se
14 fim função
```

A função *procuraVizinhosProximoNivel* utiliza a função auxiliar *vizinhosProximoNivel* para formar células que colidem com a célula de interesse em alguma direção. No teste de mesa foi descrito um bom exemplo que mostra que para a célula (1,1,2) da Figura 17, teria como célula vizinha a célula (2,1,2), porém, a célula (2,1,2) não existe, então a mesma é expandida para as células do próximo nível para se procurar pelas células que estão a direita de (1,1,2), que são as células (4,2,3) e (4,3,3). A função é capaz de realizar

buscas em todas as direções, mas a direção em que a busca deve ser feita é determinada na função *vizinhos* e passada para as demais funções de busca, exceto para a função *procuraVizinhosNivelAnteior*.

Quadro 23 - O método *procuraVizinhosNivelAnteior*

```
1  função procuraVizinhosNivelAnteior(célula, vizinhos)
2     se célula.l > 0 então
3        se tabela.contemCelula(célula) e !vizinhos.contem(célula) então
4           vizinhos.inserir(célula)
5        senão
6           se célula.l > 0 então
7              Célula c = retrair(célula)
8              procuraVizinhosNivelAnteior(c, vizinhos)
9           fim se
10       fim se
11    fim se
12 fim função
```

A função *procuraVizinhosNivelAnteior* realiza o processo de buscas de células que estão em níveis anteriores ao nível da célula de interesse. Esse processo é necessário pois uma malha pode ter células de níveis mais próximos do nível máximo que colidam com células de níveis mais próximos de 1. É o que acontece na Figura 17, onde a célula (4,3,3) tem como vizinha a célula (1,1,2).

Ao se passar uma célula para a função, é verificado se a célula existe na tabela de espalhamento. Caso exista, é inserida na tabela de espalhamento. Caso não exista, a célula é retraída e a função é chamada novamente tendo como parâmetro a célula obtida no processo de retração. Dessa forma é garantido que será obtida uma célula que está em um nível anterior e é uma vizinha da célula de interesse.

Nesta função não é necessário utilizar o parâmetro *direção*, pois a célula que é obtida no processo de retração é uma célula que está na mesma direção que a célula passada como parâmetro. Por exemplo, na Figura 17 a célula (1,1,2) está à esquerda da célula (4,3,3), mas caso a célula (1,1,2) não existisse, a célula que ocuparia o seu lugar no nível 1 é a célula (0,0,1) que também está a esquerda de (4,3,3).

Quadro 24 - O método *vizinhosProximoNivel*

```
1  função vizinhosProximoNivel(célula, direção)
2     x = 2 * célula.x
3     y = 2 * célula.y
4     l = célula.l + 1
5     Célula c1, c2
```

```
6      trocar direção
7         caso 1:
8            c1 = {x, y, l, falso}
9            c2 = {x + 1, y + 1, l, falso}
10        fim caso
11
12        caso 2:
13           c1 = {x + 1, y, l, falso}
14           c2 = {x + 1, y + 1, l, falso}
15        fim caso
16
17        caso 3:
18           c1 = {x, y, l, falso}
19           c2 = {x, y + 1, l, falso}
20        fim caso
21
22        caso 4:
23           c1 = {x, y, l, falso}
24           c2 = {x + 1, y, l, falso}
25        fim caso
26
27        caso 5:
28           c1 = {x + 1, y + 1, l, falso}
29        fim caso
30
31        caso 6:
32           c1 = {x + 1, y, l, falso}
33        fim caso
34
35        caso 7:
36           c1 = {x, y + 1, l, falso}
37        fim caso
38
39        caso 8:
40           c1 = {x, y, l, falso}
41        fim caso
42     fim troca
43     retorna Lista{c1, c2}
44  fim função
```

A função *vizinhosProximoNivel* é uma função auxiliar para a busca de células em níveis posteriores. É nesta função que são formadas as células que colidem em alguma direção com a célula de interesse. O processo de formação destas células é simples:

1. São calculadas novas coordenadas *x* e *y* multiplicando-se as coordenadas da célula passada no parâmetro da função por 2.

2. O novo nível será um incremento do nível anterior.

3. Após se obter os novos índices, é feita a análise do parâmetro *direção*, que determina qual cálculo será feito para formar as células.

O algoritmo funciona desta forma pois a célula que é passada como parâmetro será expandida para formar as células vizinhas que existem no próximo nível e que colidem com a célula de interesse. O algoritmo produz duas células para as direções esquerda, direita, acima e abaixo, e produz uma célula para as direções diagonais. O processo ocorre dessa forma pois nas diagonais sempre existirá uma única célula que será a vizinha, e nas demais direções existem duas.

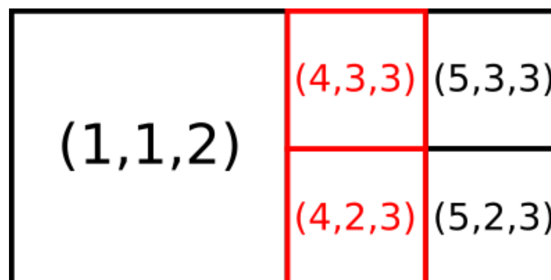


Figura 18 - Um exemplo da formação dos vizinhos no próximo nível.

Na Figura 18, é apresentado um pequeno exemplo da formação de células do próximo nível e que estão a direita da célula de interesse, que no caso é a célula (1,1,2). Na mesma região em que estão as 4 células do nível três, estaria alocada a célula (2,1,2), mas a mesma não existe na malha. Então quando a busca por vizinhos é feita, a função *vizinhosProximoNivel* realiza operações de expansão sobre a célula (2,1,2) para se obter as vizinhas corretas.

Nota-se que as células (4,2,3) e (4,3,3) embora estejam a direita da célula (1,1,2), estão posicionadas no canto esquerdo da região que teria a célula (2,1,2). Esse é outro motivo para o qual a direção de busca deve ser especificada, pois é necessário saber se as células que serão formadas colidem com a célula de interesse. Para o exemplo da Figura 18, a direção a ser informada é a 3, para pegar as células da posição esquerda da região de (2,1,2) pois estas estarão a direita de (1,1,2).

O Quadro 25 indica as posições de formação e busca de células para cada valor numérico que é passado no parâmetro *direção*:

Quadro 25 - Posições para a formação de células de acordo com cada direção

Número da Direção	**Posição na região da célula expandida**
1	Abaixo
2	Direita
3	Esquerda
4	Acima
5	Diagonal Direita Superior
6	Diagonal Direita Inferior
7	Diagonal Esquerda Superior
8	Diagonal Esquerda Inferior

No Quadro 25, a posição na região da célula expandida se trata da posição onde as células se encontram dentro da região. Por exemplo, as células (5,2,3) e (5,3,3) estão no canto direito da região de (2,1,2). As células nas diagonais seguem o mesmo padrão, porém, é necessário formar apenas uma célula para a posição diagonal. Por exemplo, na Figura 17 a célula (4,2,3) está na diagonal direita da célula (1,0,2), mas dentro da sua região ela está na posição diagonal esquerda inferior.

Cada uma das direções é definida no método *vizinhos*, sendo selecionadas de acordo com as condições definidas no método.

No pseudocódigo do Quadro 24, foi utilizada uma estrutura *trocar-caso* para realizar os cálculos de acordo com a direção solicitada. Em termos de programação o correspondente é a estrutura de seleção *switch-case*, mas poderia também ser usado *if-then-else* para a implementação.

9.3.5 Revisão Resumida da Ideia Geral Para a Busca de Vizinhanças

A ideia geral para se encontrar as célula vizinhas de uma célula de interesse é, primeiramente, aplicar as operações necessárias para se obter as células que estão aos arredores da célula de interesse no mesmo nível. Caso alguma das células calculadas não exista na tabela de espalhamento, o algoritmo começa a realizar operações recursivas até encontrar a célula que será a vizinha definitiva. Esta célula deve estar presente na tabela de espalhamento, sendo esta a regra de parada do algoritmo pois o mesmo é recursivo. E a célula encontrada pode estar em um nível anterior ou sucessor ao nível da célula de interesse.

Caso o algoritmo esteja buscando células em níveis superiores, é necessário ser informada a direção de busca, pois é utilizado um método para a geração de células do nível superior que necessita retornar as células nas posições que colidem com a célula de interesse.

Para a busca em níveis anteriores, não é necessário ter a informação da direção, pois a célula gerada pelo processo de retração estará na mesma direção que a célula que foi retraída. Lembrando que o processo de retração é utilizado pois é desta forma que é obtida a célula do nível anterior.

Esta é a visão geral para a busca de células vizinhas, onde em malhas uniformes a procura é facilitada pois não existem células distribuídas em níveis diferentes. E é devido ao fato da buscar por vizinhos também ser realizada em malhas não-uniformes, que o algoritmo acaba tendo uma complexidade mais elevada, pois é necessário contar com buscas recursivas para obter o resultado final.

10 OPERAÇÕES SOB AS CÉLULAS DA MALHA

Nesta sessão serão descritas duas operações que trabalham sob as células da malha num todo, onde todas as operações realizam modificações na malha operando as células na tabela de espalhamento. Cada operação citada corresponde a um método que a realiza na classe da malha, incluindo também as operações para o cálculo das vizinhanças que já foi descrita em uma sessão própria por se tratar de uma operação de maior importância.

10.1 ESPALHAR

A operação espalhar se trata do preenchimento uniforme de células em algum nível da malha. O seu funcionamento é o mesmo que o descrito na sessão 8.1 do experimento para a formação de células. O próprio pseudocódigo descrito no Quadro 15 é aplicado no método que implementa essa operação, mas o nível da célula não será mais uma constante inteira, pois este é fornecido para o método para se saber em qual nível espalhar as células.

10.2 NOVA CÉLULA

Esta operação é responsável por inserir uma célula na malha de forma livre, sem seguir um padrão de formação ou a regra de espalhamento. O funcionamento desta operação consiste em basicamente inserir a célula na tabela de espalhamento, pois como a inserção ocorre de forma mais direta não é necessário realizar qualquer computação sob a célula antes de inseri-la na tabela.

Quadro 26 - O pseudocódigo para a inserção de uma nova célula na malha

1	**função** novaCelula(índiceX, índiceY, índiceL)
2	tabela.inserir({índiceX, índiceY, índiceL, falso})
3	**fim função**

Lembrando que o valor *falso* que é alocado na quarta posição da formação da célula se refere ao valor booleano que diz se a célula é nula ou não.

10.3 REFINAR CÉLULA

A operação refinar célula se trata de uma operação para transformar uma célula da malha em quatro novas células do nível superior, porém, esta operação fará alterações na tabela de espalhamento. A função *expandir* descrita no tópico 9.2.1 é uma função auxiliar

aplicada para a formação de células vizinhas, mas que também é aplicada nesta operação pois é esta função que gera as quatro células necessárias.

O funcionamento da operação é simples. Dada uma célula que se deseja refinar, basta obter-se as células correspondentes do próximo nível e inseri-las na tabela de espalhamento. A célula que foi passada para ser refinada é removida da tabela, pois as células geradas ocuparão o seu lugar na malha. É por esse funcionamento que a malha e a tabela de espalhamento recebem modificações no seu conteúdo.

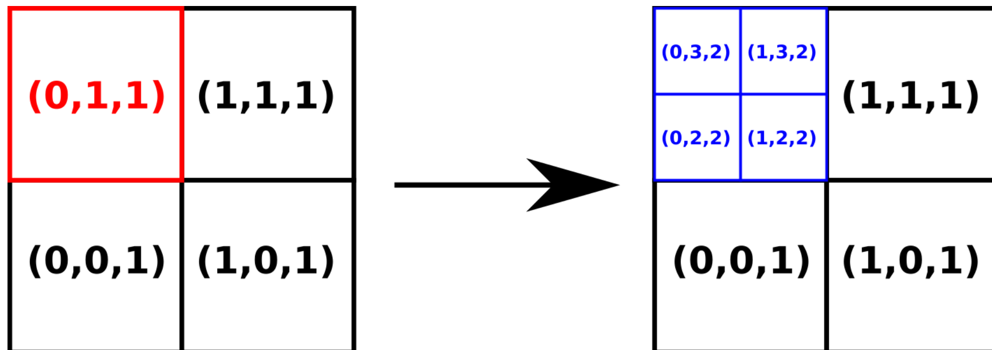


Figura 19 - Uma malha de nível 1 com refinamento na célula (0,1,1).

A Figura 19 apresenta uma malha uniforme de nível 1 que tem a célula (0,1,1) refinada, gerando uma malha não uniforme de nível 2. As células em azul na nova malha obtida refletem todo o comportamento descrito anteriormente.

Quadro 27 - Pseudocódigo do método *refinarCelula*

1	**função** refinarCelula(índiceX, índiceY, índiceL)
2	**se** tabela.contemCelula(índiceX, índiceY, índiceL) **então**
3	Lista novasCelulas = expandir({índiceX, índiceY, índiceL})
4	tabela.remover(índiceX, índiceY, índiceL)
5	tabela.inserir(novasCelulas[0])
6	tabela.inserir(novasCelulas[1])
7	tabela.inserir(novasCelulas[2])
8	tabela.inserir(novasCelulas[3])
9	**fim se**
10	**fim função**

A lista da linha 3 é uma lista de células, que obtida do método auxiliar expandir. Esta lista contém as células que foram geradas, e que são adicionadas na tabela de espalhamento através do método *inserir* que é adaptado para inserir células formadas. O método remover utilizado utilizou apenas os índices da célula de interesse pois não se tem a necessidade de usar a célula inteira para este fim.

10.4 REMOVER CÉLULA

A operação para a remoção de uma célula da malha é a mais complexa. Além da remoção da própria célula, suas células vizinhas que ocupam a mesma região devem ser removidas também, para que não exista um buraco na malha. Após todas as células envolvidas serem removidas, a célula de interesse (que é a célula passada ao método da operação) é retraída para obter-se uma célula que ocupara o lugar das células removidas.

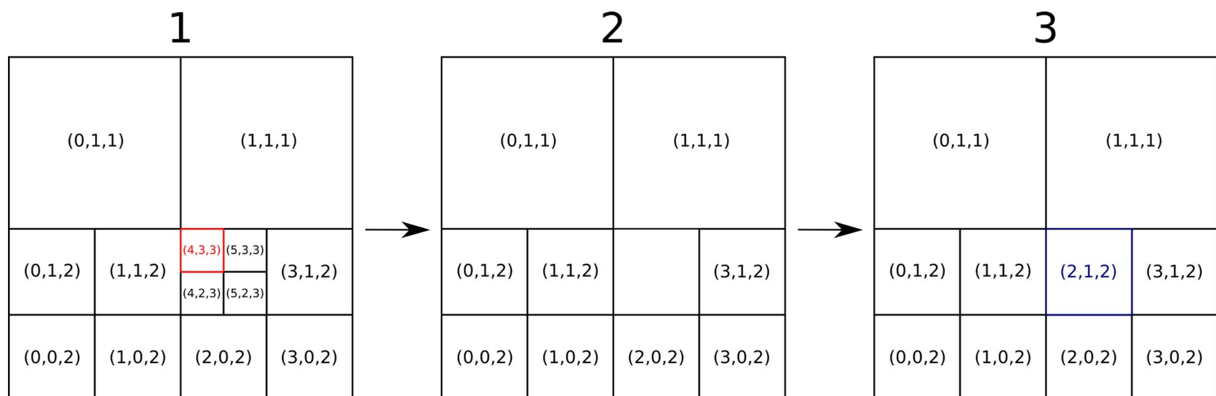


Figura 20 - As modificações realizadas na malha durante a remoção de uma célula.

A Figura 20 ilustra o processo processo de remoção com as alterações que a malha recebe. Inicialmente, tem-se interesse em remover a célula (4,3,3) em vermelho. Para realizar a remoção, são calculadas as células vizinhas que estão na mesma região de onde seria uma célula de nível 2 (que no caso é a (2,1,2)). Após estas células serem obtidas, todas as células em questão são removidas, deixando um buraco na malha. Como não é correto manter uma região sem qualquer célula, a célula (4,3,3) é retraída para se obter a correspondente no nível 2. Após aplicar a operação de retração (descrita no tópico 9.2.2), é obtida a célula (2,1,2) que é adicionada na malha e passa a ocupar o lugar das células removidas.

O cálculo de células vizinhas ocorre da mesma forma que os métodos já apresentados na sessão anterior, porém, devem ser procurados vizinhos que estão numa mesma região, e não todos os vizinhos da célula a ser removida. Então, a procura por células vizinhas é feita sem usar os métodos desenvolvidos anteriormente pois não é necessário procurar células vizinhas em um nível anterior. No entanto pode ser necessário remover células de níveis superiores e que estejam na mesma região que a célula que será removida, o que gera um processo recursivo para a remoção de células. É o caso para a remoção da célula (3,1,2), pois na mesma região existem quatro células do nível 3. A Figura 21 ilustra esta situação.

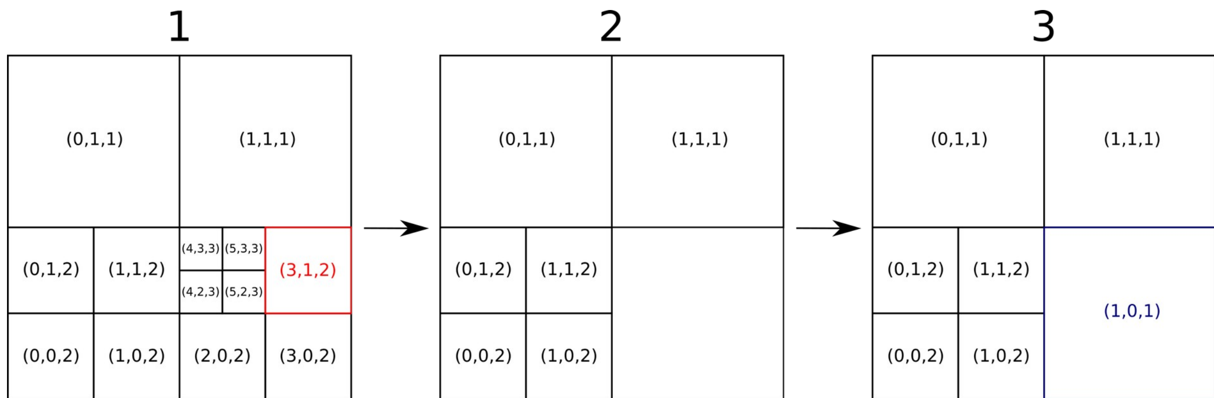


Figura 21 - Um exemplo de remoção de células com um processo recursivo.

Ao se remover a célula (3,1,2), é necessário encontrar as vizinhas que ocupem a mesma região no nível 1. Porém, a célula (2,1,2) que será calculada não está presente na malha, então, esta mesma célula é expandida para as suas correspondentes do nível superior para que também sejam removidas da malha. Esse processo é repetido cada vez que for encontrada uma célula que não está na malha. Quando a ultima célula que está na malha for removida, o processo é encerrado.

As próximas tabelas apresentam o pseudocódigo dos métodos envolvidos na remoção de células da malha.

Quadro 28 - Pseudocódigo do método *removerCelula*

1	**função** removerCelula(índiceX, índiceY, índiceL)
2	**se** tabela.contemCelula(índiceX, índiceY, índiceL) **então**
3	Célula vizinho1
4	Célula vizinho2
5	Célula vizinho3
6	**se** índiceX % 2 == 0 e índiceY % 2 == 0 **então**
7	vizinho1 = {índiceX, índiceY + 1, índiceL}
8	vizinho2 = {índiceX + 1, índiceY, índiceL}
9	vizinho3 = {índiceX + 1, índiceY + 1, índiceL}
10	**senão se** índiceX % 2 == 0 e y % 2 != 0 **então**
11	vizinho1 = {índiceX, índiceY - 1, índiceL}
12	vizinho2 = {índiceX + 1, índiceY, índiceL}
13	vizinho3 = {índiceX + 1, índiceY - 1, índiceL}
14	**senão se** índiceX % 2 != 0 e y % 2 == 0 **então**
15	vizinho1 = {índiceX, índiceY + 1, índiceL}
16	vizinho2 = {índiceX - 1, índiceY, índiceL}
17	vizinho3 = {índiceX - 1, índiceY + 1, índiceL}
18	**senão se** índiceX % 2 != 0 e y % 2 != 0 **então**
19	vizinho1 = {índiceX - 1, índiceY, índiceL}
20	vizinho2 = {índiceX, índiceY - 1, índiceL}
21	vizinho3 = {índiceX - 1, índiceY - 1, índiceL}
22	**fim se**

23	tabela.remover(índiceX, índiceY, índiceL)
24	removerVizinho(vizinho1)
25	removerVizinho(vizinho2)
26	removerVizinho(vizinho3)
27	Célula novaCélula = retrair({índiceX, índiceY, índiceL, falso})
28	tabela.inserir(novaCélula);
29	**fim se**
30	**fim função**

No Quadro 28 é apresentado o código da função principal que realiza o reconhecimento das células vizinhas existentes na mesma região, e faz as remoções necessárias. A célula de interesse que passada por parâmetro através de seus índices é removida da tabela, e as células vizinhas são removidas por meio da função auxiliar *removerVizinho*, que deve verificar se a célula passada como sendo uma suposta vizinha é uma vizinha de fato para então removê-la da tabela de espalhamento. Após todas as remoções serem feitas, a célula de interesse é retraída e a célula obtida neste processo é inserida na tabela de espalhamento.

As células vizinhas são calculadas dentro da estrutura de verificação que está entre as linhas 6 e 22. As células obtidas nestes cálculos são de fato as células que estão na mesma região que a célula de interesse. Isso é garantido pois as verificações de paridade feitas nos índices indicam em qual direção a célula está posicionada, e quando esta informação é conhecida os cálculos são feitos de forma a obter exatamente as células vizinhas que compõem a mesma região. Lembrando que após todas as células serem obtidas, o método auxiliar *removerVizinho* realiza as demais operações sob as células vizinhas.

Quadro 29 - Pseudocódigo do método *removerVizinho*

1	**função** removerVizinho(célula)
2	**se** tabela.contemCelula(célula) **então**
3	tabela.remover(célula)
4	**senão**
5	removerCelulaProximoNivel(célula)
6	**fim se**
7	**fim função**

A função *removerVizinho* é auxiliar para o processo de remoção de células. É esta função que decide se a célula já pode ser removida ou se é necessário realizar uma remoção em próximo nível. Essa decisão é tomada verificando se a célula existe na tabela de espalhamento. Caso a célula exista, ela é removida. Caso não exista é chamada a outra função auxiliar *removerCelulaProximoNivel* onde é feita a remoção em próximos níveis.

Quadro 30 - Pseudocódigo da função *removerCelulaProximoNivel*

1	**função** removerCelulaProximoNivel(célula)
2	**para** Célula c **em** expandir(célula) **faça**
3	**se** tabela.contemCelula(c) **então**
4	tabela.remover(c)
5	**senão**
6	removerCelulaProximoNivel(c)
7	**fim se**
8	**fim para**
9	**fim função**

A função *removerCelulaProximoNivel* é uma função auxiliar do processo de remoção de células da malha que realiza a remoção de células que se encontram dentro de uma mesma região mas que estão em níveis superiores. Como mostrado no exemplo da Figura 21. Nesta função é necessário expandir a célula que é passada como parâmetro pois no método *removerVizinho* foi constatado que a mesma não existe na tabela, então são obtidas as células do próximo nível para continuar a operação. As células do próximo nível são obtidas com o método *expandir* e a partir das células contidas na lista retornada pelo método, é verificado se a célula existe na tabela de espalhamento. Se sim, a célula é removida da tabela. Se não, o método *removerCelulaProximoNivel* é chamado novamente passando como parâmetro a célula não encontrada na tabela. Dessa forma ocorre o processo de remoção recursiva de células.

11 CONCLUSÃO

O objetivo principal deste trabalho era encontrar uma boa metodologia para o armazenamento de dados em uma estrutura de dados, tendo operações de manipulação dos mesmos que consumiriam tempo constante. A partir desta premissa, a estrutura de dados escolhida foi a tabela de espalhamento, pois com esta estrutura, é possível realizar o acesso aos dados por meio de um índice, possibilitando a construção das operações essenciais em tempo O(1).

Tendo a estrutura de dados, foi necessário realizar um estudo sobre os métodos numéricos que proporcionariam o melhor espalhamento de dados possível, sem colisões, e com uma complexidade em O(1). A partir disso, foram realizadas diversas implementações e modificações no código e na estrutura para se alcançar o espalhamento ideal. Graças a esse processo, foi possível observar diversos comportamentos apresentados pela estrutura ao se aplicar os testes necessários. Os comportamentos observados referem-se à disposição que os dados obtiveram na tabela ao se aplicar alguma função de espalhamento, a ocorrência de colisões na tabela, mostrando que modificações seriam necessárias, e as modificações propriamente ditas, já que houve um tempo considerável até atingir o resultado desejado. Toda essa experiencia foi crucial para se chegar no resultado obtido, que agradou não só pela sua eficiência como também pela facilidade em aplicar-se na tabela de espalhamento.

Uma das partes mais importantes do trabalho foi a construção das funções de espalhamento, que foram desenvolvidas seguindo a literatura disponível. Dentre todas as funções que foram analisadas, o destaque principal fica com a função de espalhamento perfeito, pois a forma como este método funciona foi de extrema importância para a construção da solução final. A seção de revisão do funcionamento da estratégia de espalhamento mostrou o porque desta função ter sido tão bem aplicada ao trabalho.

Para se testar as implementações que eram desenvolvidas, foram construídos experimentos de geração e espalhamento de células na tabela de espalhamento. Esses experimentos foram importantes para se observar se as abordagens seguidas seriam suficientes.

Embora exista uma estratégia geral para a implementação do que vem a ser uma função de espalhamento perfeita, a definição do espalhamento perfeito propõe apenas que os dados sejam alocados na tabela sem a ocorrência de colisões, que geram perdas. Depois de muitas implementações, e a realização de todos os testes, o objetivo de se ter uma estruturas de dados eficiente, com uma função de espalhamento perfeita foi enfim alcançado.

Referente às operações da malha, os resultados obtidos também foram satisfatórios, com destaque para a operação de busca de células vizinhas. As outras operações, para remoção e refinamento de células também foram muito bem definidas e suas aplicações são essenciais para se operar as células da malha em um todo. O ponto interessante sobre as operações de busca de vizinhanças, remoção de células, e refinamento de células, é que elas utilizam as funções auxiliares de expansão e retração de células, sendo estas muito úteis principalmente na busca de vizinhanças.

Como a tabela de espalhamento oferece operações de interação com tempo O(1), a manipulação das células nas operações da malha é facilitada, já que não é necessário perder tempo computacional e de execução com operações mais convencionais como as de estruturas de dados mais simples. Este ponto também é interessante, já que as operações em malha tem uma complexidade mais elevada, e uma estrutura que consegue lidar de forma eficiente com a manipulação dos dados ajuda no desenvolvimento destas operações.

Conclui-se então que o bom funcionamento das ideias apresentadas neste trabalho, vem da junção entre uma estrutura de dados eficiente, e a aplicação de operações que conseguem lidar com grandes quantidades de dados.

REFERÊNCIAS

CORMEN, Thomas H. et al. **Algoritmos: Teoria e Prática – 3ª edição.** Rio de Janeiro: Elsevier Editora Ltda, 2012.

SZWARCFITER, Jayme Luiz; MARKENZON Lilian. **Estruturas de Dados e Seus Algoritmos – 3ª edição.** Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora Ltda, 2010.

FORTUNA, Armando de Oliveira. **Técnicas Computacionais para Dinâmica dos Fluidos - Conceitos Básicos e Aplicações.** São Paulo: Edusp - Editora da Univcrsidade de São Paulo, 2000.

BENTLEY, Jon Louis. **Multidimensional Binary Search Trees Used for Associative Searching.** Communications of the ACM, ACM New York, NY, USA, 1975.

BERG, Mark de. et al. **Computational Geometry – Algorithms and Applications, third edition.** Springer-Verlag Berlin Heidelberg, 2008.

CPLUSPLUS. **vector::operator[] – C++ Reference.** Disponível em: <https://www.cplusplus.com/reference/vector/vector/operator%5B%5D/>. Acesso em: 28/02/2022.

ANEXO A – Artigo

Geração de Malha Adaptativas Usando Tabelas de Espalhamento

Thiago M. Salvador[1], Álvaro J. P. Franco[2]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC, Brasil.

thiago.martendal@grad.ufsc.br

Abstract. *This work presents the construction of hash tables with numerical algorithms and data structures, for the representation and generation of adaptive meshes with refinement. Meshes are configured as a network of cells, where cells are the basic component of the mesh. Each mesh cell is stored in the hash table following a hash function defined by a numerical operation. The hash table must offer basic operations for interacting with the data in O(1) time, as a mesh can generate millions of cells, which makes it necessary to define efficient operations to deal with voluminous amounts of data. For example, the analysis of temperature in an environment can generate meshes with a large number of cells.*

Resumo. *Este trabalho apresenta a construção de tabelas de espalhamento com algoritmos numéricos e de estruturas de dados, para a representação e geração de malhas adaptativas com refinamento. As malhas são configuradas como uma rede de células, onde as células são o componente básico da malha. Cada célula da malha é armazenada na tabela de espalhamento seguindo uma função de espalhamento definida por uma operação numérica. A tabela de espalhamento deve oferecer operações básicas de interação com os dados em tempo O(1), pois uma malha pode gerar milhões de células, o que torna necessário a definição de operações eficientes para lidar com volumosas quantidades de dados. Por exemplo, a análise de temperatura em um ambiente, pode gerar malhas com uma grande quantidade de células.*

1. Objetivos

O objetivo deste trabalho é construir estruturas para a representação de malhas e células, e uma tabela de espalhamento que tenha a aplicação de uma função de espalhamento que armazene os dados sem a ocorrência de colisões. Com a proposta de se construir uma tabela de espalhamento que garanta o melhor resultado para o armazenamento de dados através de funções numéricas, também surge a necessidade de se ter operações de interação com os dados para esta tabela com complexidade O(1). Após a conclusão da tabela de espalhamento, será construída a malha, que se trata de uma estrutura intermediária que gerencia as operações de tabela e o tratamento de células em um todo.

2. Uma Malha

Uma malha se trata de uma rede de células que podem se expandir até determinado nível. Este nível deve ser determinado antes da construção da malha e é conhecido como o nível máximo da malha. A malha tem esse comportamento para tornar possível a discretização de algum fenômeno ou problema, buscando sua solução ao mesmo tempo que pode ser possível criar uma representação gráfica.

A ideia de se dividir a malha em níveis tem como objetivo tornar acessível uma região em questão através dos identificadores das células, o que permite se trabalhar partes específicas da malha sem criar modificações em um domínio.

Desta forma, é possível operar problemas por partes que são descritas em diferentes regiões do domínio, contribuindo para uma solução final.

Toda a malha é descrita de um nível de 0 a n, com $n > 0$. O nível zero de uma malha se trata do nível base, que compõem a região total da malha. Uma malha pode ter um grande número de níveis, mas neste trabalho foram realizados testes de implementação considerando 12 como sendo o nível máximo da malha, pois na prática não serão tratadas malhas com níveis maiores.

A Figura 1 mostra um pequeno exemplo de uma malha de nível 2 com 16 células (pois $4^2 = 16$), sem índices em suas células.

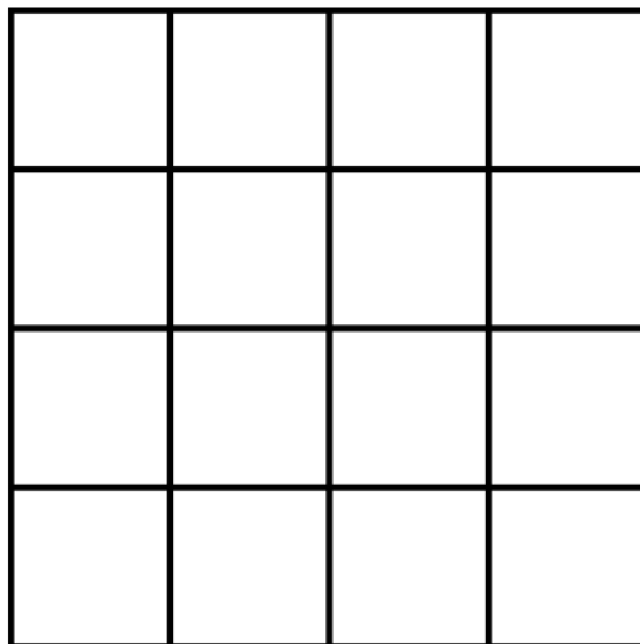


Figura 1. Uma malha de nível 2 sem índices.

2.1. Representação de Uma Malha

Uma malha pode ser representada com uma classe que contém uma estrutura de dados capacitada a armazenar as células (a tabela de espalhamento) e métodos que implementam as operações necessárias para a interação com as células.

As operações existentes na classe da malha não devem ser confundidas com as operações da tabela de espalhamento, pois a malha é pensada para se olhar as células como um todo, mesmo que estas possam ser operadas de forma individual. Já as operações da tabela se referem apenas as questões de armazenamento e resgate das células.

Apesar de ser interessante ter uma classe específica para a malha, é possível realizar operações em outros escopos de algum programa, basta se obter a tabela que armazena as células e então é possível programar novas operações além da classe especificada. E isso é importante para as futuras aplicações deste trabalho, pois os analistas numéricos podem construir suas próprias operações e métodos que a malha geral talvez não ofereça.

Para este trabalho, portanto, as operações necessárias a malha serão descritas na classe da malha.

3. Uma Célula

Uma célula é uma componente elementar de uma malha. Uma célula é identificada por 3 valores inteiros: $c = (x, y, l)$, onde c identifica a célula, x é o índice da coordenada no eixo das abscissas, y é o índice da coordena no eixo das ordenadas, e l é o nível da célula, que é obtido a partir do processo de refinamento. Nenhum dos três valores de indexação são negativos e todos eles são números inteiros.

Os índices da tabela são obtidos seguindo o plano cartesiano 2D.

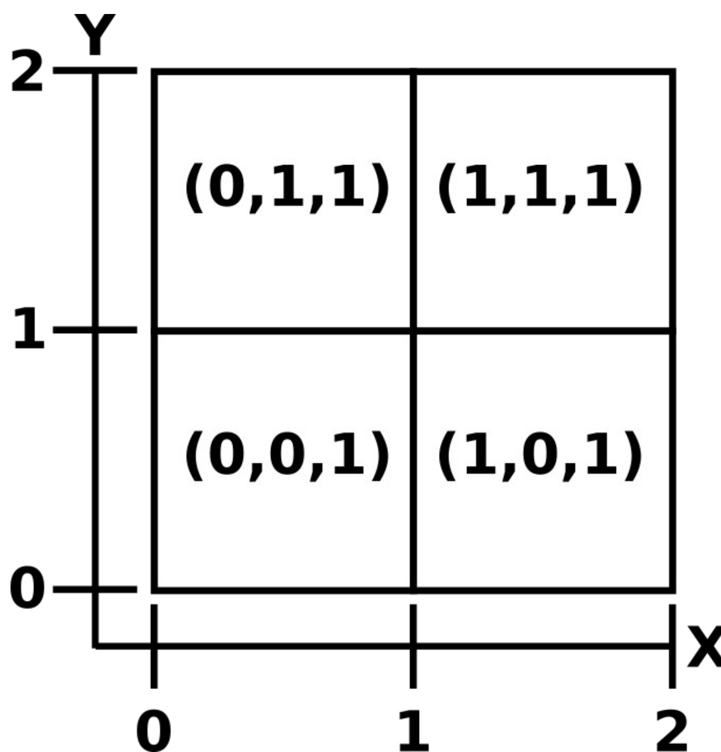


Figura 2. Uma malha de nível 1 com 4 células.

A Figura 2 mostra o esquema de uma malha uniforme, onde todas as células se dividem de forma igual até atingir o nível em questão.

Como as células são classificadas em níveis, elas possuem espessura. Uma célula é dita pertencer ao nível mais fino se ela está no nível máximo permitido pela malha. Uma célula que está no primeiro nível, pertence ao nível mais grosso. Portanto, os níveis de cada célula representam sua espessura até o nível máximo da malha.

No caso de uma malha não-uniforme, pode-se ter a ocorrência de células de níveis diferentes, como mostra a Figura 3.

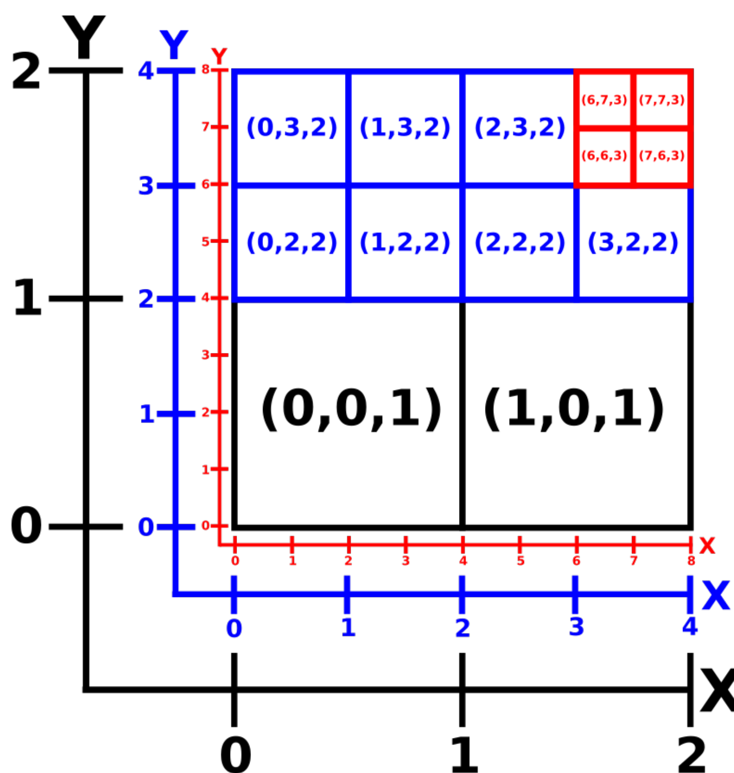


Figura 3. Malha não-uniforme de nível 3.

Na Figura 3 foi representada uma malha não-uniforme de nível 3. Cada nível de células existente na malha foi representado em cores diferentes, para mostrar como pode ocorrer a disposição de células em uma malha não-uniforme. Para cada nível existente, a formação de índices das células segue um plano cartesiano específico, que será detalhado mais adiante. As células coloridas em preto pertencem ao nível mais grosso, e as células coloridas em vermelho pertencem ao nível mais fino. É importante ressaltar que o crescimento dos índices de células ocorre de acordo com o plano cartesiano em cada nível, logo, conforme os níveis avançam, o plano cartesiano é ajustado para atender o número total de células existentes naquele nível.

Cada célula em uma malha pode ser dividida em no máximo 4 células, logo, basta calcular para se descobrir o número total de células em um nível da malha, onde *l* representa o nível da célula. Este processo é denominado refinamento de célula.

Dada toda a descrição de como uma célula é composta, e como ocorre seu processo de refinamento, é definido que a única célula do nível 0 é (0, 0, 0), pois $4^0 = 1$. A célula do nível 0 cobre todo o domínio e representa o nível mais grosso da malha.

3.1. Representação de uma Célula

Neste trabalho, uma célula contém apenas seus valores de indexação, pois o foco não é fazer a aplicação das malhas, mas apenas a construção de estruturas que as representem.

Portanto, uma célula pode ser representada com uma estrutura, que se trata de um artifício computacional para agrupar dados (em algumas linguagens de programação uma estrutura é definida com o comando *struct*). Porém pode-se haver a necessidade de se conter mais dados além dos índices, e as células podem necessitar de operações específicas. Então, é possível representar uma célula com uma classe para estes casos, devido à facilidade de se construir um componente mais completo.

Para este trabalho, a célula será representada com uma estrutura que contém os 3 índices, e um parâmetro *nulo* para indicar se a célula está nula ou não, sendo esta representação suficiente para o desenvolvimento do trabalho. Também fica convencionado que os valores *x*, *y* e *l* são inicializados com 0, e o parâmetro nulo é inicializado com o valor *true* (verdadeiro), indicando que a célula está inicialmente nula. Quando as células forem inicializadas, os valores são alterados.

Quadro 1. O Pseudocódigo da estrutura de célula

1	**estrutura** Célula
2	x = 0
3	y = 0
4	l = 0
5	nulo = verdadeiro
6	**fim estrutura**

O Quadro 1 apresenta o pseudocódigo da estrutura de célula descrita neste tópico.

4. Uma Tabela de Espalhamento

A tabela de espalhamento é uma estrutura de dados que realiza o mapeamento de dados para uma determina posição através de um índice, que é calculado a partir de uma função de espalhamento. O dado quando atribuído a um índice é chamado de dado satélite. A função de espalhamento tem por objetivo calcular um índice através de parâmetros fornecidos para algum método numérico que realizará o cálculo. Após este processo, é retornado o índice em questão que representa o local na tabela de espalhamento onde o dado será alocado. Esta configuração dá a ideia da existência de um par entre índice e dado satélite, como ocorre em estruturas como mapas e dicionários, porém, em tabelas de espalhamento a ocorrência de pares não é totalmente concreta, pois dados diferentes podem ser alocados para um mesmo índice, gerando uma colisão.

As colisões se tratam da disputa de dois ou mais dados satélites diferentes por um mesmo local na tabela de espalhamento. Essa disputa ocorre quando o método numérico escolhido para gerar os índices, gera índices repetidos para dados diferentes, mesmo que estes dados não tenham parâmetros repetidos. Portanto, deve-se avaliar maneiras de se lidar com a ocorrência de colisões no processo de espalhamento. A forma mais tradicional para impedir colisões é utilizar listas encadeadas nas posições da

tabela de espalhamento, assim, quando uma colisão ocorrer, os dados serão armazenados na lista que está na posição selecionada. Esse método para a solução de colisões é chamado de encadeamento. Embora este processo evite a perda de dados, não foi aplicado neste trabalho, pois não permite a construção de operações com baixa complexidade já que para encontrar determinado valor na lista seria necessário fazer um percurso sobre a mesma.

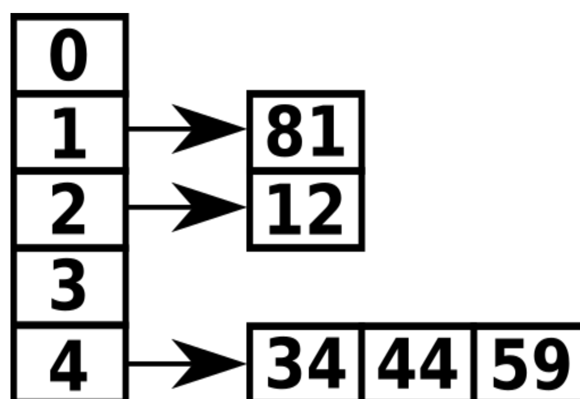


Figura 4. Exemplo de tabela de espalhamento.

A Figura 4 mostra uma tabela de espalhamento com capacidade para se inserir até 5 elementos, tendo ocorrido colisões na posição 4 da tabela.

É usado um vetor ou uma lista que dispõem de uma lista encadeada em cada posição para se resolver o problema das colisões. Caso não fosse utilizado nenhum método para tratar as colisões, alguns dados seriam perdidos. Neste exemplo, se as colisões não fossem tratadas, o valor 34 seria substituído pelo valor 44, que seria substituído pelo valor 59.

5. Funções de Espalhamento

Uma função de espalhamento – também conhecida como função de dispersão ou função hash – se trata de uma função que mapeia uma chave intermediária para uma determinada posição da tabela de espalhamento, como mencionado anteriormente. Essa função é calculada através de uma operação numérica que utiliza determinados parâmetros incluindo a chave de uma célula. Abaixo serão apresentadas algumas funções de espalhamento que foram consideradas para implementações anteriores da tabela de espalhamento, juntamente com a função de espalhamento escolhida para a implementação final.

5.1. A Chave Intermediária

Para se calcular o espalhamento com as funções numéricas, é necessário ter uma chave que representa o valor a ser inserido na tabela de espalhamento. Para uma tabela que armazena valores inteiros, a chave intermediária pode ser o próprio valor a ser inserido, já que se trata de um inteiro. Porém, nem sempre existe a conveniência de se armazenar apenas valores numéricos na tabela. Para o caso da inserção de valores não-numéricos, o dado a ser inserido pode ser acompanhado de um parâmetro que indica qual é o valor

da chave intermediária, ou a chave pode ser obtida a partir de operações numéricas que são feitas utilizando parâmetros ou propriedades existentes no dado a ser inserido. Sendo assim, abaixo seguem as operações aplicadas para se obter a chave intermediária:

1. Número total de colunas no nível da célula $= 2^{\text{Nível da célula}}$
2. Chave $=$ (Número total de colunas no nível da célula $* \ y) + x$

A chave obtida a partir deste cálculo pode ser utilizada como chave intermediária para ser aplicada em uma função de espalhamento, mas também pode ser diretamente aplicada como uma posição para armazenar um elemento na tabela de espalhamento, sem a necessidade de se usar uma função de espalhamento mais específica, ou seja, um endereçamento direto com a chave intermediária. Isso é possível devido à forma como o espalhamento perfeito foi aplicado na tabela de espalhamento, que será discutido nas próximas seções.

5.2. A Função de Espalhamento com o Método da Divisão

O espalhamento com o método da divisão se trata de um cálculo extremamente simples para se descobrir o índice onde um dado será inserido na tabela de espalhamento.

Para se obter o índice do dado basta apenas calcular o resto da divisão entre a chave intermediária e o tamanho que foi alocado para a tabela de espalhamento.

Portanto:

$$h(k) = k \% m$$

Onde *k* é a chave intermediária, *m* é o tamanho da tabela, e *h(k)* é a aplicação da função de espalhamento.

5.3. A Função de Espalhamento Perfeito

Um dos objetivos deste trabalho é encontrar a função de espalhamento perfeita, que consegue atribuir posições específicas para os dados na tabela de espalhamento sem a ocorrência de colisões. Quando uma função de espalhamento consegue atender este requisito, a função é perfeita, por isso recebe o nome de função de espalhamento perfeito.

Uma forma de se implementar a função de espalhamento perfeita é descrita a seguir:

- Uma tabela de espalhamento em dois níveis é construída. O primeiro nível se trata da tabela principal, e o segundo nível se trata de uma segunda tabela que deve armazenar os dados.

- Ambas as tabelas podem ser representadas com uma lista ou com vetores.

- Em cada um dos níveis é aplicado uma função de espalhamento para determinar a posição onde o dado será alocado na lista ou no vetor.

- A tabela do primeiro nível terá como tamanho a quantidade de dados que serão inseridos, e a tabela de segundo nível terá como tamanho o total de dados que se espalham naquele nível elevado ao quadrado. Por exemplo, se na primeira posição da tabela de primeiro nível for inserido 4 elementos, a tabela de segundo nível tera como tamanho $4^2 = 16$.

Os passos acima descrevem uma abordagem tradicional para se desenvolver o método de espalhamento perfeito, e é necessário ter cautela em sua utilização, pois uma quantidade de dados absurdamente grande pode ocasionar problemas de memória, podendo interromper a execução de um programa que utiliza este método.

Devido aos resultados obtidos com a aplicação da adaptação do método de transposição e o método da divisão para o segundo nível, não é necessário que as tabelas de segundo nível tenham como tamanho o quadrado do número de valores que se espalham na posição onde a tabela de segundo nível está, pois foi desenvolvida uma nova abordagem para realizar o espalhamento perfeito. A abordagem desenvolvida propõe utilizar o valor do nível máximo da malha + 1 como sendo o tamanho da tabela de primeiro nível, e para as tabelas de segundo nível, o tamanho será 4 elevado a posição onde a tabela de segundo nível se encontra na tabela de primeiro nível. Essa abordagem é possível pois a adaptação do método de transposição e o método da divisão conseguem alocar os dados de forma perfeita com as tabelas tendo os tamanhos descritos. E também não é necessário se preocupar com problemas de memória, pois neste trabalho é definido que uma malha terá como nível máximo o nível 12, sendo assim, a tabela de espalhamento armazenará 16.777.216 células, sendo um número de dados que a tabela de espalhamento consegue armazenar sem problema algum.

A Figura 5 mostra o esquema de uma tabela de espalhamento com o método de espalhamento perfeito em dois níveis, onde a tabela de primeiro nível armazena as tabelas de cada nível da malha, e as tabelas do segundo nível armazenam as células de cada nível da malha. Cada posição das tabela é numerada com o seu índice, partindo de 0.

Figura 5. Tabela de espalhamento em dois níveis.

Na Figura 5, em vermelho está a tabela de primeiro nível, e em azul estão as tabelas do segundo nível. Para a tabela da posição 0 é alocado um único elemento pois $4^0 = 1$, para a tabela da posição 1 tem-se $4^1 = 4$, e para a tabela da posição 2 o tamanho alocado é $4^2 = 16$.

A respeito dos níveis da tabela e o nível da malha, são conceitos diferentes e não devem ser confundidos. Os níveis da tabela se referem as distribuições das listas que formam a tabela de espalhamento, como na Figura 5, onde em vermelho está a tabela de primeiro nível, e em azul estão as tabelas do segundo nível. E o nível da malha se refere a um valor numérico que determina a quantidade de células que podem ser alocadas na malha, por exemplo, uma malha uniforme de nível 2 terá $4^2 = 16$ células.

Tendo-se o esquema de agrupamento de tabelas definido, o Quadro 2 mostra a quantidade de células que cada nível de malha possui.

Quadro 2. A quantidade de células por nível

Nível da Malha	Total de Células	Total de Células ao Quadrado
0	1	1
1	4	16
2	16	256
3	64	4096
4	256	65536
5	1024	1048576
6	4096	16777216
7	16384	268435456
8	65536	4294967296
9	262144	68719476736
10	1048576	$1,099511628\times10^{12}$
11	4194304	$1,759218604\times10^{13}$
12	16777216	$2,814749767\times10^{14}$

Estes valores foram obtidos seguindo o cálculo para se determinar a quantidade máxima de células em cada nível, que é 4^l onde l é o nível da malha atual.

5.4. As Funções de Espalhamento Escolhidas

Foram realizados testes de aplicação dos métodos de espalhamento na tabela, onde a chave intermediária e o método da divisão conseguiram garantir um bom espalhamento sem colisões e com um tempo computacional O(1). Portanto, a aplicação destes dois métodos consegue satisfazer a necessidade de se ter o espalhamento perfeito.

5.5. A Implementação das Funções Escolhidas

Ainda neste tópico, já foram apresentados os métodos numéricos de espalhamento, agora será detalhada a implementação destes métodos com seus respectivos pseudocódigos e as complexidades de cada função.

5.5.1. O Cálculo da Chave Intermediária

No Quadro 3 segue o pseudocódigo para o cálculo da chave intermediária com a complexidade de cada instrução:

Quadro 3. Pseudocódigo para o cálculo da chave intermediária

	Código	Complexidade
1	**função** calculoChave(índiceX, índiceY, nívelL)	
2	colunasNível = 2^nívelL	O(1)
3	chave = (colunasNível * índiceY) + índiceX	O(1)
4	**retorna** chave	
5	**fim função**	

5.5.2. O Método da Divisão

No Quadro 4 segue o pseudocódigo do método da divisão e a complexidade de cada instrução:

Quadro 4. Pseudocódigo do método da divisão

Código	Função
h = calculoChave(índiceX, índiceY, nívelL) % tamanhoTabela	O(1)

Como definido anteriormente, o método *calculoChave* tem tempo O(1), e a operação de resto da divisão também é definida com tempo O(1), portanto, o método da divisão também tem tempo O(1). A variável *tamanhoTabela* representa o tamanho total da tabela de segundo nível, que foi determinado previamente e nunca é alterado. Então, a quantidade de elementos em uma tabela não altera a complexidade total.

6. A Implementação da Tabela de Espalhamento

Após as devidas definições, foi feita a implementação da tabela de espalhamento. Para representar a tabela e suas operações, foi feita uma classe que contém os métodos de espalhamento selecionados, as operações de inserção, remoção e busca, e as estruturas que armazenam os dados.

6.1. O Armazenamento dos Dados

Os dados são armazenados em um vetor, porém, como está sendo aplicado o espalhamento perfeito, é necessário representar a tabela em dois níveis. Isso é feito utilizando um vetor de vetores. Dessa forma, o vetor do primeiro nível armazena os vetores do segundo nível. Os vetores do segundo nível armazenam os dados que são espalhados.

Os vetores são representadas com a biblioteca vector do C++, sendo assim, os níveis da tabela são representados como um vector de vectors (vetor de vetores) na implementação.

Quadro 5. Um vetor de vetores com a estrutura vector

```
std::vector<std::vector<Celula>> tabela;
```

A estrutura no Quadro 5 terá como tamanho no primeiro nível, o nível total da malha, e no segundo nível, o tamanho será 4 elevado ao índice atual da malha que vai de 0 ao nível total da malha. E este tamanho deve ser previamente inicializado, e isso é feito com um laço de repetição que varia de $i = 0$ até o nível total da malha. Então, a tabela da posição i recebe como tamanho 4^i . Este processo é necessário devido a requisição do tamanho total das tabelas de segundo nível para a aplicação das funções de espalhamento.

6.2. A Implementação das Operações

6.2.1. Definições

Todas as operações descritas nesta seção terão a representação em pseudocódigo para se ter um embasamento de como cada operação funciona. Lembrando que a tabela de espalhamento possui dois níveis, e para acessar o primeiro nível é utilizado o nível da célula que é representado pela variável *índiceL*.

A variável *índiceH* é utilizada para armazenar o valor do índice da célula na tabela de segundo nível, e está utilizando o método da divisão para calcular o índice com a chave intermediária, assim, esta variável está presente nos pseudocódigos de todas as operações aqui descritas.

O método de cálculo de chave intermediária já foi descrito e demonstrado que sua complexidade é O(1).

6.2.2. Inserção

A operação de inserção fica responsável por inserir um elemento na tabela, verificando a ocorrência de uma colisão. A inserção é feita recebendo uma célula e utilizando seus índices para se calcular o índice da tabela de espalhamento. Após o cálculo de índice ser efetuado, é verificado se a posição na tabela de segundo nível contém uma célula nula ou não. Esta verificação é feita analisando se o parâmetro *nulo* é verdadeiro ou falso. Se *nulo* for verdadeiro, não houve colisão pois a posição da tabela contém uma célula nula, e a nova célula é inserida em seu lugar. Caso contrário, houve uma colisão e uma mensagem de erro é reportada interrompendo a execução do programa.

No Quadro 6 segue a representação do método de inserção em pseudocódigo, com as complexidades de cada operação:

Quadro 6. Pseudocódigo do método de inserção

1	**função** inserir(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**se** tabela[índiceL][índiceH].l != -1 **então**	O(1)
4	Houve uma colisão e interrompe a execução	O(1)
5	**fim se**	
6	tabela[índiceL][índiceH] = {índiceX, índiceY, índiceL}	O(1)
7	**fim função**	

Como todas as operações da função tem complexidade constante, a operação de inserção é realizada em tempo O(1).

A verificação realizada na linha 3 é feita por segurança, caso uma célula que já está presente na tabela de espalhamento seja inserida novamente por algum descuido, será detectada uma colisão. Caso ocorra uma colisão, a instrução da linha 6 não será executada pois o programa é interrompido. Também vale lembrar que para a tabela de primeiro nível não é aplicado nenhum método de espalhamento, pois o espalhamento para o primeiro nível é feito de forma direta utilizando o índice *l*.

6.2.3. Remoção

A operação de remoção realiza a substituição de uma célula em uma posição da tabela por uma célula nula. Essa operação é feita dessa forma pois não deve ser reduzido o tamanho total da tabela, e nenhuma posição deve deixar de existir, pois outro elemento pode ser inserido em seu lugar.

Para realizar a remoção, também é calculado o índice da célula na tabela de espalhamento, que utiliza os parâmetros que estão na célula que foi passada ao método de remoção. Após o índice ser calculado, a célula na posição do índice na tabela de segundo nível é substituída pela célula nula. A célula nula é definida com seus três índices em 0 e o parâmetro nulo em verdadeiro.

No Quadro 7 segue a representação do método de remoção em pseudocódigo com sua complexidade:

Quadro 7. Pseudocódigo do método de remoção

1	**função** remover(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	tabela[índiceL][índiceH] = {0, 0, 0, verdadeiro}	O(1)
4	**fim função**	

6.2.4. Busca

A operação de busca retorna uma célula da tabela de espalhamento, de acordo com os índices passados ao método. Então, basta calcular o índice da célula e retorná-la ao fim do método. Caso não exista nenhuma célula na posição, é retornada uma célula nula.

No Quadro 8 segue a representação do método de busca em pseudocódigo com sua complexidade:

Quadro 8. Pseudocódigo do método de busca

1	**função** buscaCelula(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**retorna** tabela[índiceL][índiceH]	O(1)
4	**fim função**	

6.2.5. Contém Célula

A operação contém célula deve indicar se a tabela contém uma célula ou não. Para o método é passado os 3 valores de indexação da célula, e então é calculado índice da célula na tabela. Ao ter o índice definido, basta comparar se os valores *x*, *y* e *l* da célula na tabela são iguais aos valores *x*, *y* e *l* passados ao método. Se sim, o método retorna verdadeiro, senão retorna falso. Esta operação é empregada para verificar se uma célula que foi inserida na tabela está de fato na tabela.

No Quadro 9 segue a representação do método contém célula em pseudocódigo com sua complexidade:

Quadro 9. Pseudocódigo do método contém célula

1	**função** contemCelula(índiceX, índiceY, índiceL)	
2	índiceH = calculoChave(índiceX, índiceY, índiceL) % tamanho da tabela de segundo nível	O(1)
3	**retorna** ((tabela[índiceL][índiceH].x == x) e (tabela[índiceL][índiceH].y == y) e (tabela[índiceL][índiceH].l == l))	O(1)
4	**fim função**	

7. Experimentos e Resultados

7.1. Experimento - A Formação de Células

O uso real das malhas terão as células fornecidas pela aplicação em que a tabela de espalhamento será aplicada, portanto, para se testar as implementações construídas, era necessário produzir as células que seriam inseridas na tabela. E isso foi feito com implementação de uma estratégia para gerar as células necessárias para os testes. A partir dessa necessidade, foi feito este experimento que utiliza um algoritmo que espalha as células até um determinado nível.

Como descrito em seções anteriores, uma célula é definida por $c = (x, y, l)$, onde x, y e l indexam a célula, sendo l o nível ao qual a célula pertence. O nível da célula é definido previamente. Quando se deseja criar uma malha de nível único, o valor do nível é constante. Quando a malha deve armazenar células de mais de um nível, o nível deve variar em um intervalo que inicia no nível mínimo até o nível máximo que se deseja representar. O intervalo é feito com o percurso de um laço de repetição do nível mínimo até o nível máximo. Ambos os valores de nível devem ser positivos, e o nível mínimo pode iniciar em 0.

Para definir os índices x e y, também é feito um percurso através de um laço de repetição, iniciando em 0 até 4^l para garantir o número total de células em cada nível. Dentro do laço de repetição, o índice y é incrementado, e quando o valor de y for igual a $\sqrt{4^l}$ o índice x é incrementado e o índice y é zerado.

Os índices são incrementados desta forma pois é necessário garantir que não serão gerados índices que não correspondem ao nível atual.

Como um pequeno exemplo, dada uma malha uniforme de nível 2, o número de células geradas será $4^2 = 16$ que também é o tamanho da tabela de segundo nível da posição 2. Porém os índices x e y não serão incrementados até 16, pois quando o valor de y for igual a $\sqrt{4^2} = 4$ o índice x é incrementado e o índice y é zerado, portanto, os índices x e y crescem no intervalo [0,3] no nível 2.

No Quadro 10 segue um pseudocódigo que apresenta as configurações descritas acima:

Quadro 10. Pseudocódigo para o experimento de formação de células

```
1  indiceX = 0
2  indiceY = 0
3  nivelAtual = 2
4  para k de 0 ate 4 ^ nivelAtual faça
5      célula = {indiceX, indiceY, nivelAtual}
6      indiceY++
7      se indiceY == ((4 ^ nivelAtual) ^ (½)) então
8        indiceY = 0
9        indiceX++
10     fim se
11 fim para
```

7.2. Experimento - Geração Aleatória de Células

Após a construção da tabela de espalhamento, foi realizado um experimento para se gerar 3.000.000 (três milhões) de células de forma totalmente aleatória. Para isso, foi utilizada a função *rand* da biblioteca <cstdlib> para se gerar números aleatórios dentro de laços de repetição que garantissem que o número total de células seria gerado.

No Quadro 11 segue o pseudocódigo da implementação deste experimento:

Quadro 11. Pseudocódigo do experimento de geração aleatória de células

```
1  numeroDeCelulas = 3000000
2  nivelMaximo = 11
3  tabela = Tabela(nivelMaximo)
4  enquanto tabela.getTotalElementos() < numeroDeCelulas faça
5     para i de 0 até numeroDeCelulas faça
6        se tabela.totalDeElementos() == numeroDeCelulas então
7           fim para
8        fim se
9        índiceL = rand() % nivelMaximo+1
10       fator = pow(2, l)
11       índiceX = rand() % fator
12       índiceY = rand() % fator
13       se tabela.contemCelula({índiceX, índiceY, índiceL}) == falso então
14          tabela.inserir({índiceX, índiceY, índiceL})
15       fim se
16    fim para
17 fim enquanto
```

Como o número de células a ser gerado é muito grande, foi necessário preparar a tabela para alocar células até o nível 11.

Embora este experimento seja simples, foi importante para poder se visualizar o comportamento da geração de células em meio a imposição de uma geração randômica. E devido ao fato do número de células geradas ser bastante alto, a malha para armazenar as células seria não-uniforme e de tamanho $4^{11} = 4194304$.

É importante destacar que o método *contemCelula* na linha 13 está verificando se a célula já foi inserida na tabela, e essa verificação é necessária pois se for passada uma célula que já está na tabela para o método *inserir*, vai ser reportada uma colisão e o experimento falha. E esta verificação também é necessária pois como as células estão sendo geradas de forma aleatória, uma mesma célula pode ser gerada diversas vezes.

Outra observação importante refere-se a função *rand()*, que se trata da função para a geração de número aleatórios.

7.3. Revisão do Funcionamento da Estratégia de Espalhamento

Durante o processo de implementação da tabela de espalhamento, foram feitas muitas análises a respeito dos métodos de espalhamento que seriam aplicados. A princípio, o método de espalhamento universal apresentava o resultado mais aceitável para se

aplicar na tabela. Porém, este método tinha a desvantagem de se necessitar realizar um espalhamento prévio apenas para se ter conhecimento de onde os elementos seriam alocados, e não era possível espalhar os dados sem realizar este processo, pois a função de espalhamento universal não fornecia nenhuma estratégia de alocação sem o conhecimento prévio dos dados.

É importante lembrar que o espalhamento dos dados ocorre de forma muito melhor se os mesmos forem previamente conhecidos, pois é possível aplicar estratégias como o espalhamento prévio para se determinar as posições finais dos dados, no entanto, após muitos testes e verificações, foi encontrada a abordagem perfeita de espalhamento.

Com o uso do próprio nível da célula para realizar o endereçamento direto no primeiro nível, foi necessário aplicar uma função de espalhamento apenas no segundo nível. As funções escolhidas foram uma adaptação do método de transposição de células aplicado de forma direta, e o método da divisão. Ambos os métodos apresentaram resultados satisfatórios, portanto, os dois são aptos para realizar o espalhamento, mas vale lembrar que a adaptação do método de transposição realiza o cálculo da chave intermediária.

Tendo sido escolhidas as funções para aplicar na tabela de espalhamento, foram analisados os resultados de inserção de células, e foi observado que a abordagem aplicada consegue manter as células da malha ordenadas na tabela, e isso é muito interessante pois os dados podem ser organizados de forma padrão dentro da tabela, e essa organização é definida em ordem crescente dos índices da célula, como mostra a Figura 6:

Malha de nível 1:
[1][0] = {0,0,1}
[1][1] = {1,0,1}
[1][2] = {0,1,1}
[1][3] = {1,1,1}

Figura 6. Um exemplo de como os dados são dispostos com o espalhamento perfeito.

Para uma malha de nível 1, as 4 células possíveis da malha estão espalhadas de forma crescente, sendo as células ordenadas primeiramente de acordo com o índice x, e logo após pelo índice y. E esse ordenamento ocorre nas tabelas de segundo nível, pois as tabelas de primeiro nível são ordenadas pelos níveis das células, ou seja, a célula do nível 0 é a primeira célula da tabela, logo após são inseridas as células do nível 1, do nível 2, e assim por diante.

Após concluir todas as análises, observar os resultados e tendo sida construída uma tabela de espalhamento com operações que respeitam o tempo de complexidade constante, é enfim concluída a pesquisa por uma função de espalhamento com baixa complexidade e sem a ocorrência de colisões, ou seja, uma função de espalhamento perfeita. Mas não foi encontrado apenas uma função, e sim toda uma metodologia que soluciona a questão principal deste trabalho.

8. As vizinhanças

Agora será tratada a questão das vizinhanças em uma malha, que se refere ao processo de busca pelas células vizinhas de uma célula atual c.

8.1. Operações Necessárias Para a Busca de Células Vizinhas

Antes de entrar na estratégia de busca de células vizinhas, é importante destacar duas operações essenciais para este processo, que são as operações de expansão e retração de uma célula.

8.1.1. A Expansão de uma Célula

A operação de expansão de uma célula é o processo de se expandir uma célula de um nível l em quatro novas células do próximo nível $(l+1)$. Este processo é feito a partir dos parâmetros (x,y,l) existentes na célula. Portanto, para uma célula $c = (x,y,l)$, o algoritmo realiza os seguintes passos:

1. São obtidos novos parâmetros $x1$, $y1$ e $l1$, onde:

- $x1 = 2 \times x$

- $y1 = 2 \times y$
- $l1 = l + 1$

2. São formadas as quatro novas células {*c1*, *c2*, *c3*, *c4*} a partir de *c*, que terão os seguintes valores:

   - *c1 = (x1, y1, l1)*
   - *c2 = (x1, y1+1, l1)*
   - *c3 = (x1+1, y1, l1)*
   - *c4 = (x1+1, y1+1, l1)*

3. Após a formação das novas células, as mesmas são retornadas pelo método em uma lista.

Em um pequeno exemplo, tem-se interesse em expandir a célula *c = (0,1,1)*. A partir de *c*, são formadas as células {*c1*, *c2*, *c3*, *c4*}:

1. São obtidos os novos parâmetros:

   - $x1 = 2 \times 0 = 0$
   - $y1 = 2 \times 1 = 2$
   - $l1 = 1 + 1 = 2$

2. São formadas as quatro novas células:

   - *c1 = (0, 2, 2)*
   - *c2 = (0, 2+1, 2) = (0, 3, 2)*
   - *c3 = (0+1, 2, 2) = (1, 2, 2)*
   - *c4 = (0+1, 2+1, 2) = (1, 3, 2)*

3. Ao final é retornada a lista com as novas células.

Quadro 12. O pseudocódigo para a expansão de uma célula

```
1  função expandir(célula)
2    x = 2 * célula.x
3    y = 2 * célula.y
4    l = célula.l + 1
5    Célula c1 = {x, y, l}
6    Célula c2 = {x, y + 1, l}
7    Célula c3 = {x + 1, y, l}
8    Célula c4 = {x + 1, y + 1, l}
9    retorna Lista{c1, c2, c3, c4}
10 fim função
```

8.1.2. A Retração de uma Célula

A operação de retração de uma célula segue o processo contrário à operação de expansão. A partir de uma célula *c = (x,y,l)*, tem-se interesse em descobrir uma célula *c1 = (x1,y1,l1)* que ocupa a região de c no nível anterior. Para isso, são realizados os seguintes passos:

1. Se x for par:
   - $x1 = x \div 2$
2. Se x for impar:
   - $x1 = (x - 1) \div 2$
3. Se y for par:
   - $y1 = y \div 2$
4. Se y for impar:
   - $y1 = (y - 1) \div 2$
5. O nível l1 é o decremento de l:
   - $l1 = l - 1$
6. Ao executar todos os passos, a célula c1 é retornada pelo método.

Para um exemplo, será aplicado o algoritmo de retração sob a célula *c = (0,3,2)*:

1. 0 é par:
   - $x1 = 0 \div 2 = 0$
2. 3 é ímpar:
   - $y1 = (3-1) \div 2 = 1$
3. O novo nível é obtido:
   - $l1 = 2 - 1 = 1$
4. É retornada a célula *c1 = (0,1,1)*.

Quadro 13. O pseudocódigo para a retração de uma célula

```
1 função retrair(célula)
2   x = ((célula.x % 2 == 0) ? célula.x / 2 : (célula.x - 1) / 2)
3   y = ((célula.y % 2 == 0) ? célula.y / 2 : (célula.y - 1) / 2)
4   l = célula.l - 1
5   retorna Célula{x, y, l}
6 fim função
```

Nas linhas 2 e 3 foram utilizados operadores ternários para decidir qual cálculo deve ser feito de acordo com a paridade do índice.

8.1.3. Observações

Estes algoritmos são necessários para as operações de busca de células vizinhas, pois caso não exista alguma célula na tabela de espalhamento que deveria ser a célula vizinha, esta célula que não foi encontrada será expandida e retraída até se encontrar as células que colidem com a célula de interesse para se determinar as vizinhanças.

Estes algoritmos não geram alterações na malha e na tabela de espalhamento, apenas realizam suas devidas funções e retornam as células desejadas. Porém, outros

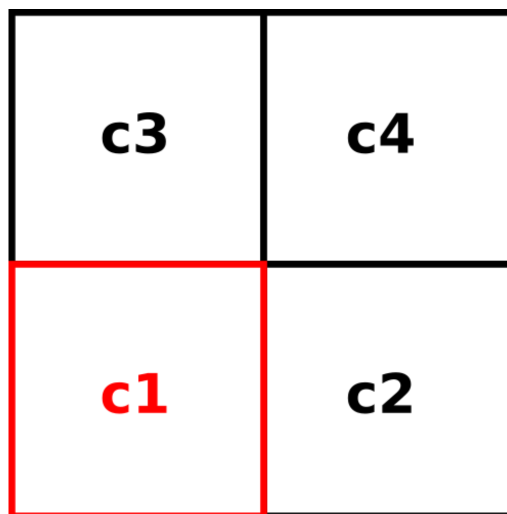
métodos que serão apresentados nos próximos capítulos fazem uso destes algoritmos para realizar modificações na malha para determinado interesse. Então, estes algoritmos assumem papeis auxiliares importantes para operações em malha.

8.2. A Estratégia de Busca de Células Vizinhas

Dada uma célula $c = (x,y,l)$, para se encontrar as células vizinhas de c, deve ser feita uma busca em dois passos na malha. O primeiro passo consiste em avaliar as células vizinhas da célula *c* e que estão no mesmo nível da malha. Caso não exista alguma ou nenhuma célula que colida com c no mesmo nível, o algoritmo realiza o segundo passo de busca onde é feita uma busca recursiva até ser encontrada uma célula que colida com *c* (esta célula pode estar em um nível de malha anterior ou próximo ao nível de *c*).

8.2.1. O Primeiro Passo de Busca

A busca no primeiro passo é feita de forma simples, pois basta aplicar operações aritméticas básicas nos índices de c para formar as células que colidem com c no mesmo nível de malha, e caso estas células estejam incluídas na tabela de espalhamento significa que elas existem naquele nível de malha, portanto são adicionadas na lista de células vizinhas de *c*. A Figura 7 ilustra as operações realizadas para se encontrar as células vizinhas no mesmo nível de malha.



$$c1 = (x, y, l)$$

$$c2 = (x+1, y, l)$$

$$c3 = (x, y+1, l)$$

$$c4 = (x+1, y+1, l)$$

Figura 7. Aplicação da busca de células vizinhas em uma malha uniforme de nível 1.

Na Figura 7, a célula *c1* é destacada em vermelho pois se trata da célula que se deseja buscar os vizinhos. As células *c2*, *c3* e *c4* são as células vizinhas pois estas colidem com a célula *c1*. Para a busca ser realizada, as células *c2*, *c3* e *c4* são formadas a partir dos índices *(x,y,l)* pertencentes a célula *c1*, e estes índices são também coloridos em vermelho para indicar sua origem.

A regra de formação das células é muito clara, como *c2* é uma célula que está a direita de *c1*, basta somar a coordena *x* de *c1* com 1 para se obter a coordenada *x* de *c2*. Já para *c4*, são somadas as duas coordenadas pois c4 está na diagonal superior direita. Caso existisse uma célula a esquerda de *c1*, a coordenada *x* desta célula seria formada subtraindo a coordenada *x* de *c1* por 1. Como a malha do exemplo é de nível 1, não foi necessário realizar operações de subtração, mas para malhas de níveis maiores onde existem mais células, são necessárias as aplicações das operações de adição e subtração para se formar todas as células vizinhas. A Figura 15 abaixo apresenta um exemplo de busca de uma célula em uma malha de nível 2.

c4	c5	c6	
c2	c1	c3	
c7	c8	c9	

$c1 = (x, y, l)$ $c6 = (x+1, y+1, l)$

$c2 = (x-1, y, l)$ $c7 = (x-1, y-1, l)$

$c3 = (x+1, y, l)$ $c8 = (x, y-1, l)$

$c4 = (x-1, y+1, l)$ $c9 = (x+1, y-1, l)$

$c5 = (x, y+1, l)$

Figura 8. A busca de células vizinhas para uma malha uniforme de nível 2.

Na Figura 8, a célula *c1* é a célula de interesse para se encontrar as células vizinhas, e como *c1* se encontra em uma região central da malha de nível 2, são necessárias mais operações para se encontrar todas as células que colidem com *c1*.

Este primeiro passo de busca é suficiente para se encontrar todas as células vizinhas em uma malha uniforme, mas caso alguma das células formadas não estivesse na tabela de espalhamento, significaria que a célula não está presente na malha, portanto, a malha seria não-uniforme, e isso leva ao segundo passo de busca que trata da busca por células em uma vizinhança de malha não-uniforme.

8.2.2. O Segundo Passo de Busca

O segundo passo de busca é ativado quando deve-se procurar as células vizinhas em uma malha não-uniforme. Caso alguma célula formada no primeiro passo de busca não exista na tabela de espalhamento, e por consequência, na malha, esta célula que não existe é expandida para as células do próximo nível, e então é verificado se as células que colidem com a célula de busca atual existem na tabela de espalhamento, e caso existam, estas células são adicionadas a lista de células vizinhas, caso não existam, o processo é repetido recursivamente passando as células que não existem para ser feita a procura no próximo nível.

O processo de busca com a expansão das células que não foram encontradas na tabela de espalhamento é feito para procurar os vizinhos no próximo nível. Mas, também deve ser feita a procura nas células do nível anterior, pois também podem existir células vizinhas no nível anterior. Então, a célula que não foi encontrada no mesmo nível que a célula de interesse da busca, deve ser retraída para a célula de nível

anterior, onde será feita uma analise de colisão com as células daquele nível, para se encontrar as células vizinhas. Este processo também é recursivo.

Na subseção 8.2.4 são descritos os algoritmos para a busca das células vizinhas.

8.2.3. Teste de Mesa

A partir da Figura 9 será feito um teste de mesa onde se tem interesse em descobrir quais são as células vizinhas da célula (4,3,3):

Figura 9. Uma malha não-uniforme de nível 3 onde a célula (4,3,3) é a célula de interesse.

Para se encontrar as células vizinhas de (4,3,3), o processo é iniciado com o primeiro passo de busca tendo os seguintes resultados:

Quadro 14. Resultados do primeiro passo de busca para a malha da Figura 9

c1 = (4, 3, 3)	c6 = (4 + 1, 3 + 1, 3) = (5, 4, 3)
c2 = (4 - 1, 3, 3) = (3, 3, 3)	c7 = (4 - 1, 3 - 1, 3) = (3, 2, 3)
c3 = (4 + 1, 3, 3) = (5, 3, 3)	c8 = (4, 3 - 1, 3) = (4, 2, 3)
c4 = (4 - 1, 3 + 1, 3) = (3, 4, 3)	c9 = (4 + 1, 3 - 1, 3) = (5, 2, 3)
c5 = (4, 3 + 1, 3) = (4, 4, 3)	

Como a célula (4,3,3) está no nível mais fino da malha (ou seja, o nível máximo) não se tem a necessidade de buscar vizinhos no próximo nível, pois este não existe. Então, o algoritmo realiza a busca no nível anterior ao nível da célula (4,3,3), que é o nível 2. Para isso, o algoritmo de busca de vizinhos invoca o algoritmo de retração em cada célula que não foi encontrada no primeiro passo de busca para se descobrir qual a célula que está na região da mesma no nível anterior. Neste teste de mesa, as células encontradas no primeiro passo foram coloridas em azul, e as células não encontradas estão em preto na tabela.

Portanto, para as células (3,3,3), (3,4,3), (4,4,3) e (3,2,3), o algoritmo retrairá as células, até encontrar uma célula que exista na tabela de espalhamento, sendo esta célula uma vizinha de (4,3,3). Para as células (3,3,3) e (3,2,3), o algoritmo de retração formou a célula (1,1,2), que existe na malha e colide com a célula (4,3,3). Para a célula (3,4,3), o algoritmo formou a célula (1,2,2), que não existe na malha, então, a célula (1,2,2) é retraída formando a célula (0,1,1), que está na malha e é vizinha de (4,3,3). Por fim, para a célula (4,4,3), o algoritmo de retração formou a célula (2,2,2), que não existe na malha, então é novamente retraída, formando a célula (1,1,1), que está presente na malha e é vizinha de (4,3,3).

O algoritmo em dois passos conseguiu encontrar todas as células vizinhas de (4,3,3) em uma malha não-uniforme, como mostra a Figura 10:

(0,1,1) (1,1,1)
(0,1,2) (1,1,2) (4,3,3) (5,3,3) (4,2,3) (5,2,3) (3,1,2)
(0,0,2) (1,0,2) (2,0,2) (3,0,2)

Figura 10. As células vizinhas de (4,3,3).

Estão coloridas em azul todas as vizinhas de (4,3,3).

O outro caso interessante de busca de vizinhos de uma célula, é quando as células vizinhas estão em um nível superior a célula de interesse. Por exemplo, para a malha da Figura 10, a célula (1,1,2) teria como vizinha a direita a célula (2,1,2), porém, esta não existe na malha. Então, o segundo passo do algoritmo faria a retração e a expansão da célula (2,1,2) para encontrar as células vizinhas nos níveis anterior e posterior. Como nenhuma célula será encontrada no nível anterior para (2,1,2) o algoritmo continua apenas com as células do nível posterior. E aplicando-se a expansão da célula (2,1,2), são formadas as células (4,2,3), (4,3,3), (5,2,3) e (5,3,3). Todas estas células existem na malha, mas apenas duas delas são vizinhas de (1,1,2), e para saber quem são as vizinhas, o algoritmo escolhe as duas células que estão a direita de (1,1,2), pois (2,1,2) também estaria a direita caso existisse na malha.

8.2.4. O Pseudocódigo Para as Estratégias de Busca de Células Vizinhas

Neste tópico são apresentados os pseudocódigos dos métodos que realizam a busca de células vizinhas.

Quadro 15. A função principal para a busca de células vizinhas

```
1  função vizinhos(índiceX, índiceY, índiceL)
2     vizinhos = lista de células
3     fator = ((4 ^ índiceL) ^ (1 / 2)) - 1
4
5     se  índiceY > 0 então
6        Célula c = {índiceX, índiceY - 1, índiceL}
7        inserirVizinho(c, vizinhos, 1)
8     fim se
9
10    se índiceX > 0 então
11       Célula c = {índiceX - 1, índiceY, índiceL}
12       inserirVizinho(c, vizinhos, 2)
13    fim se
14
15    se índiceX < fator então
16       Célula c = {índiceX + 1, índiceY, índiceL}
17       inserirVizinho(c, vizinhos, 3)
18    fim se
19
20    se índiceY < fator então
21       Célula c = {índiceX, índiceY + 1, índiceL}
22       inserirVizinho(c, vizinhos, 4)
23    fim se
24
25    se índiceX > 0 e índiceY > 0 então
26       Célula c = {índiceX - 1, índiceY - 1, índiceL}
27       inserirVizinho(c, vizinhos, 5)
28    fim se
29
30    se índiceX > 0 e índiceY < fator então
31       Célula c = {índiceX - 1, índiceY + 1, índiceL}
32       inserirVizinho(c, vizinhos, 6)
33    fim se
34
35    se índiceX < fator e índiceY> 0 então
36       Célula c = {índiceX + 1, índiceY - 1, índiceL}
37       inserirVizinho(c, vizinhos, 7);
38    fim se
39
40    se índiceX < fator e índiceY < fator então
41       Célula c = {índiceX + 1, índiceY + 1, índiceL}
42       inserirVizinho(c, vizinhos, 8)
43    fim se
44
45    retorna vizinhos
46 fim função
```

Na função vizinhos é inciado o primeiro passo de busca por células vizinhas da célula de interesse. As células vizinhas são calculadas através de operações básicas e enviadas a função *inserirVizinho*.

Cada condição determina a direção em que células vizinhas deverão ser buscadas na função *inserirVizinho*, porém essa busca será realizada apena se a célula c não existir na tabela de espalhamento, e a direção só interessa para a busca de células em níveis superiores, não sendo necessária para a busca em níveis anteriores ao nível da célula de interesse.

Quadro 16. O método inserirVizinho

```
1  função inserirVizinho(célula, vizinhos, direção)
2     se tabela.contemCelula(célula) então
3        vizinhos.inserir(célula)
4     senão
5        procuraVizinhosProximoNivel(célula, vizinhos, direção)
6        procuraVizinhosNivelAnteior(célula, vizinhos)
7     fim se
8  fim função
```

A função *inserirVizinho* deve inserir a célula obtida no primeiro passo de busca na lista de vizinhos, caso esta célula exista na tabela de espalhamento. Se a célula não existir na tabela, a função inicia o segundo passo de busca chamando as funções *procuraVizinhosProximoNivel* e *procuraVizinhosNivelAnteior*. Isso acontece pois se a célula não existe na tabela de espalhamento, significa que podem existir células nos níveis anteriores e sucessores que ocupem sua região, e estas células vem a ser as células vizinhas.

Quadro 17. O método procuraVizinhosProximoNivel

```
1  função procuraVizinhosProximoNivel(célula, vizinhos, direção)
2     se célula.l < nivelMalha então
3        células = vizinhosProximoNivel(célula, direção)
4        para c em células faça
5           se tabela.contemCelula(c) e !c.nulo então
6              vizinhos.inserir(c);
7           senão
8              se c.l < nivelMalha e !c.nulo então
9                 procuraVizinhosProximoNivel(c, vizinhos, direção)
10             fim se
11          fim se
12       fim para
13    fim se
14 fim função
```

A função *procuraVizinhosProximoNivel* utiliza a função auxiliar *vizinhosProximoNivel* para formar células que colidem com a célula de interesse em

alguma direção. No teste de mesa foi descrito um bom exemplo que mostra que para a célula (1,1,2) da Figura 10, teria como célula vizinha a célula (2,1,2), porém, a célula (2,1,2) não existe, então a mesma é expandida para as células do próximo nível para se procurar pelas células que estão a direita de (1,1,2), que são as células (4,2,3) e (4,3,3). A função é capaz de realizar buscas em todas as direções, mas a direção em que a busca deve ser feita é determinada na função vizinhos e passada para as demais funções de busca, exceto para a função *procuraVizinhosNivelAnteior*.

Quadro 18. O método procuraVizinhosNivelAnteior

1	**função** procuraVizinhosNivelAnteior(célula, vizinhos)
2	**se** célula.l > 0 **então**
3	**se** tabela.contemCelula(célula) e !vizinhos.contem(célula) **então**
4	vizinhos.inserir(célula)
5	**senão**
6	**se** célula.l > 0 **então**
7	Célula c = retrair(célula)
8	procuraVizinhosNivelAnteior(c, vizinhos)
9	**fim se**
10	**fim se**
11	**fim se**
12	**fim função**

A função *procuraVizinhosNivelAnteior* realiza o processo de buscas de células que estão em níveis anteriores ao nível da célula de interesse. Esse processo é necessário pois uma malha pode ter células de níveis mais próximos do nível máximo que colidam com células de níveis mais próximos de 1. É o que acontece na Figura 10, onde a célula (4,3,3) tem como vizinha a célula (1,1,2).

Ao se passar uma célula para a função, é verificado se a célula existe na tabela de espalhamento. Caso exista, é inserida na tabela de espalhamento. Caso não exista, a célula é retraída e a função é chamada novamente tendo como parâmetro a célula obtida no processo de retração. Dessa forma é garantido que será obtida uma célula que está em um nível anterior e é uma vizinha da célula de interesse.

Nesta função não é necessário utilizar o parâmetro direção, pois a célula que é obtida no processo de retração é uma célula que está na mesma direção que a célula passada como parâmetro. Por exemplo, na Figura 10 a célula (1,1,2) está à esquerda da célula (4,3,3), mas caso a célula (1,1,2) não existisse, a célula que ocuparia o seu lugar no nível 1 é a célula (0,0,1) que também está a esquerda de (4,3,3).

Quadro 19. O método vizinhosProximoNivel

1	**função** vizinhosProximoNivel(célula, direção)
2	x = 2 * célula.x
3	y = 2 * célula.y
4	l = célula.l + 1
5	Célula c1, c2
6	**trocar** direção
7	**caso** 1:
8	c1 = {x, y, l, falso}

```
9          c2 = {x + 1, y + 1, l, falso}
10       fim caso
11
12       caso 2:
13          c1 = {x + 1, y, l, falso}
14          c2 = {x + 1, y + 1, l, falso}
15       fim caso
16
17       caso 3:
18          c1 = {x, y, l, falso}
19          c2 = {x, y + 1, l, falso}
20       fim caso
21
22       caso 4:
23          c1 = {x, y, l, falso}
24          c2 = {x + 1, y, l, falso}
25       fim caso
26
27       caso 5:
28          c1 = {x + 1, y + 1, l, falso}
29       fim caso
30
31       caso 6:
32          c1 = {x + 1, y, l, falso}
33       fim caso
34
35       caso 7:
36          c1 = {x, y + 1, l, falso}
37       fim caso
38
39       caso 8:
40          c1 = {x, y, l, falso}
41       fim caso
42    fim troca
43    retorna Lista{c1, c2}
44 fim função
```

A função *vizinhosProximoNivel* é uma função auxiliar para a busca de células em níveis posteriores. É nesta função que são formadas as células que colidem em alguma direção com a célula de interesse. O processo de formação destas células é simples:

1. São calculadas novas coordenadas *x* e *y* multiplicando-se as coordenadas da célula passada no parâmetro da função por 2.

2. O novo nível será um incremento do nível anterior.

3. Após se obter os novos índices, é feita a análise do parâmetro direção, que determina qual cálculo será feito para formar as células.

O algoritmo funciona desta forma pois a célula que é passada como parâmetro será expandida para formar as células vizinhas que existem no próximo nível e que colidem com a célula de interesse. O algoritmo produz duas células para as direções esquerda, direita, acima e abaixo, e produz uma célula para as direções diagonais. O processo ocorre dessa forma pois nas diagonais sempre existirá uma única célula que será a vizinha, e nas demais direções existem duas.

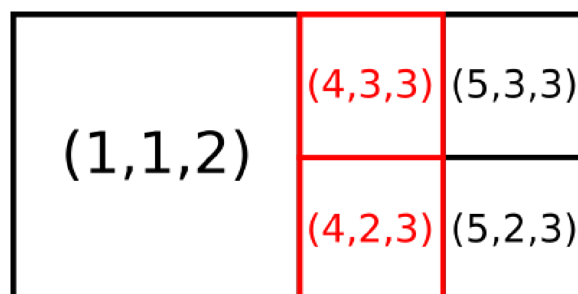


Figura 11. Um exemplo da formação dos vizinhos no próximo nível.

Na Figura 11, é apresentado um pequeno exemplo da formação de células do próximo nível e que estão a direita da célula de interesse, que no caso é a célula (1,1,2). Na mesma região em que estão as 4 células do nível três, estaria alocada a célula (2,1,2), mas a mesma não existe na malha. Então quando a busca por vizinhos é feita, a função *vizinhosProximoNivel* realiza operações de expansão sobre a célula (2,1,2) para se obter as vizinhas corretas.

Nota-se que as células (4,2,3) e (4,3,3) embora estejam a direita da célula (1,1,2), estão posicionadas no canto esquerdo da região que teria a célula (2,1,2). Esse é outro motivo para o qual a direção de busca deve ser especificada, pois é necessário saber se as células que serão formadas colidem com a célula de interesse. Para o exemplo da Figura 11, a direção a ser informada é a 3, para pegar as células da posição esquerda da região de (2,1,2) pois estas estarão a direita de (1,1,2).

O Quadro 20 indica as posições de formação e busca de células para cada valor numérico que é passado no parâmetro direção:

Quadro 20. Posições para a formação de células de acordo com cada direção

Número da Direção	Posição na região da célula expandida
1	Abaixo
2	Direita
3	Esquerda
4	Acima
5	Diagonal Direita Superior

6	Diagonal Direita Inferior
7	Diagonal Esquerda Superior
8	Diagonal Esquerda Inferior

No Quadro 20, a posição na região da célula expandida se trata da posição onde as células se encontram dentro da região. Por exemplo, as células (5,2,3) e (5,3,3) estão no canto direito da região de (2,1,2). As células nas diagonais seguem o mesmo padrão, porém, é necessário formar apenas uma célula para a posição diagonal. Por exemplo, na Figura 10 a célula (4,2,3) está na diagonal direita da célula (1,0,2), mas dentro da sua região ela está na posição diagonal esquerda inferior.

Cada uma das direções é definida no método vizinhos, sendo selecionadas de acordo com as condições definidas no método.

No pseudocódigo do Quadro 19, foi utilizada uma estrutura *trocar-caso* para realizar os cálculos de acordo com a direção solicitada. Em termos de programação o correspondente é a estrutura de seleção *switch-case*, mas poderia também ser usado *if-then-else* para a implementação.

8.2.5. Revisão Resumida da Ideia Geral Para a Busca de Vizinhanças

A ideia geral para se encontrar as célula vizinhas de uma célula de interesse é, primeiramente, aplicar as operações necessárias para se obter as células que estão aos arredores da célula de interesse no mesmo nível. Caso alguma das células calculadas não exista na tabela de espalhamento, o algoritmo começa a realizar operações recursivas até encontrar a célula que será a vizinha definitiva. Esta célula deve estar presente na tabela de espalhamento, sendo esta a regra de parada do algoritmo pois o mesmo é recursivo. E a célula encontrada pode estar em um nível anterior ou sucessor ao nível da célula de interesse.

Caso o algoritmo esteja buscando células em níveis superiores, é necessário ser informada a direção de busca, pois é utilizado um método para a geração de células do nível superior que necessita retornar as células nas posições que colidem com a célula de interesse.

Para a busca em níveis anteriores, não é necessário ter a informação da direção, pois a célula gerada pelo processo de retração estará na mesma direção que a célula que foi retraída. Lembrando que o processo de retração é utilizado pois é desta forma que é obtida a célula do nível anterior.

Esta é a visão geral para a busca de células vizinhas, onde em malhas uniformes a procura é facilitada pois não existem células distribuídas em níveis diferentes. E é devido ao fato da buscar por vizinhos também ser realizada em malhas não-uniformes, que o algoritmo acaba tendo uma complexidade mais elevada, pois é necessário contar com buscas recursivas para obter o resultado final.

9. Operações Sob As Células da Malha

Nesta sessão serão descritas duas operações que trabalham sob as células da malha num todo, onde todas as operações realizam modificações na malha operando as células na

tabela de espalhamento. Cada operação citada corresponde a um método que a realiza na classe da malha, incluindo também as operações para o cálculo das vizinhanças que já foi descrita em uma sessão própria por se tratar de uma operação de maior importância.

9.1. Espalhar

A operação espalhar se trata do preenchimento uniforme de células em algum nível da malha. O seu funcionamento é o mesmo que o descrito na sessão 7.1 do experimento para a formação de células. O próprio pseudocódigo descrito no Quadro 10 é aplicado no método que implementa essa operação, mas o nível da célula não será mais uma constante inteira, pois este é fornecido para o método para se saber em qual nível espalhar as células.

9.2. Nova Célula

Esta operação é responsável por inserir uma célula na malha de forma livre, sem seguir um padrão de formação ou a regra de espalhamento. O funcionamento desta operação consiste em basicamente inserir a célula na tabela de espalhamento, pois como a inserção ocorre de forma mais direta não é necessário realizar qualquer computação sob a célula antes de inseri-la na tabela.

Quadro 21. O pseudocódigo para a inserção de uma nova célula na malha

1	**função** novaCelula(índiceX, índiceY, índiceL)
2	tabela.inserir({índiceX, índiceY, índiceL, falso})
3	**fim função**

Lembrando que o valor falso que é alocado na quarta posição da formação da célula se refere ao valor booleano que diz se a célula é nula ou não.

9.3. Refinar Célula

A operação refinar célula se trata de uma operação para transformar uma célula da malha em quatro novas células do nível superior, porém, esta operação fará alterações na tabela de espalhamento. A função expandir descrita no tópico 8.1.1 é uma função auxiliar aplicada para a formação de células vizinhas, mas que também é aplicada nesta operação pois é esta função que gera as quatro células necessárias.

O funcionamento da operação é simples. Dada uma célula que se deseja refinar, basta obter-se as células correspondentes do próximo nível e inseri-las na tabela de espalhamento. A célula que foi passada para ser refinada é removida da tabela, pois as células geradas ocuparão o seu lugar na malha. É por esse funcionamento que a malha e a tabela de espalhamento recebem modificações no seu conteúdo.

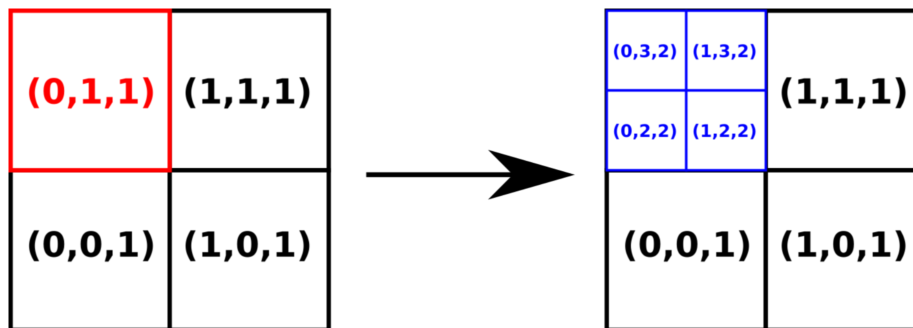


Figura 12. Uma malha de nível 1 com refinamento na célula (0,1,1).

A Figura 12 apresenta uma malha uniforme de nível 1 que tem a célula (0,1,1) refinada, gerando uma malha não uniforme de nível 2. As células em azul na nova malha obtida refletem todo o comportamento descrito anteriormente.

Quadro 22. Pseudocódigo do método refinarCelula

1	**função** refinarCelula(índiceX, índiceY, índiceL)
2	**se** tabela.contemCelula(índiceX, índiceY, índiceL) **então**
3	Lista novasCelulas = expandir({índiceX, índiceY, índiceL})
4	tabela.remover(índiceX, índiceY, índiceL)
5	tabela.inserir(novasCelulas[0])
6	tabela.inserir(novasCelulas[1])
7	tabela.inserir(novasCelulas[2])
8	tabela.inserir(novasCelulas[3])
9	**fim se**
10	**fim função**

A lista da linha 3 é uma lista de células, que obtida do método auxiliar expandir. Esta lista contém as células que foram geradas, e que são adicionadas na tabela de espalhamento através do método inserir que é adaptado para inserir células formadas. O método remover utilizado utilizou apenas os índices da célula de interesse pois não se tem a necessidade de usar a célula inteira para este fim.

9.4. Remover Célula

A operação para a remoção de uma célula da malha é a mais complexa. Além da remoção da própria célula, suas células vizinhas que ocupam a mesma região devem ser removidas também, para que não exista um buraco na malha. Após todas as células envolvidas serem removidas, a célula de interesse (que é a célula passada ao método da operação) é retraída para obter-se uma célula que ocupara o lugar das células removidas.

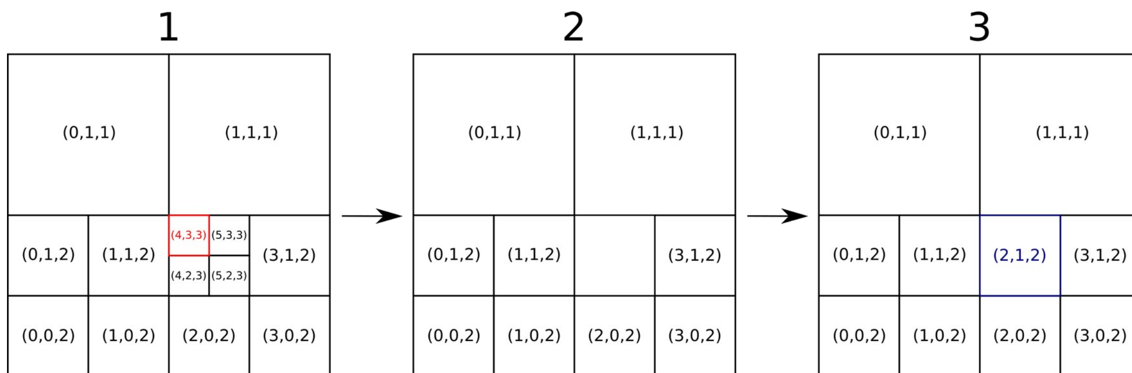


Figura 13. As modificações realizadas na malha durante a remoção de uma célula.

A Figura 13 ilustra o processo processo de remoção com as alterações que a malha recebe. Inicialmente, tem-se interesse em remover a célula (4,3,3) em vermelho. Para realizar a remoção, são calculadas as células vizinhas que estão na mesma região de onde seria uma célula de nível 2 (que no caso é a (2,1,2)). Após estas células serem obtidas, todas as células em questão são removidas, deixando um buraco na malha. Como não é correto manter uma região sem qualquer célula, a célula (4,3,3) é retraída para se obter a correspondente no nível 2. Após aplicar a operação de retração (descrita no tópico 9.2.2), é obtida a célula (2,1,2) que é adicionada na malha e passa a ocupar o lugar das células removidas.

O cálculo de células vizinhas ocorre da mesma forma que os métodos já apresentados na sessão anterior, porém, devem ser procurados vizinhos que estão numa mesma região, e não todos os vizinhos da célula a ser removida. Então, a procura por células vizinhas é feita sem usar os métodos desenvolvidos anteriormente pois não é necessário procurar células vizinhas em um nível anterior. No entanto pode ser necessário remover células de níveis superiores e que estejam na mesma região que a célula que será removida, o que gera um processo recursivo para a remoção de células. É o caso para a remoção da célula (3,1,2), pois na mesma região existem quatro células do nível 3. A Figura 14 ilustra esta situação.

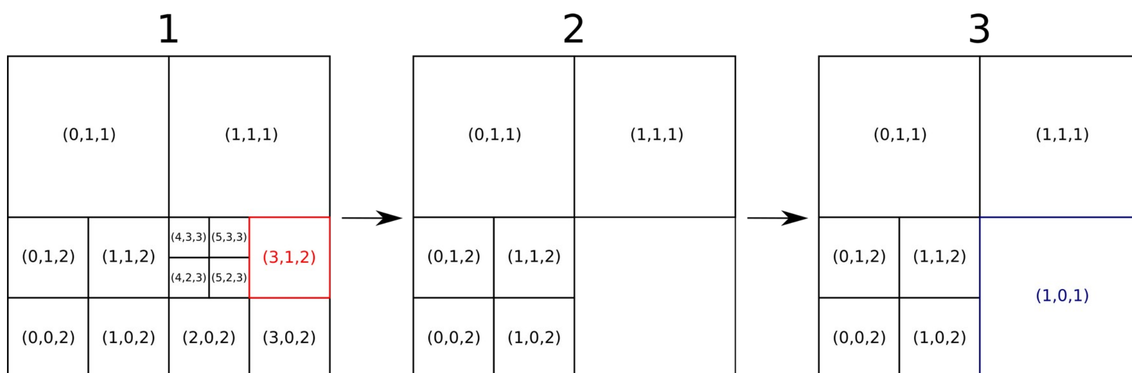


Figura 14. Um exemplo de remoção de células com um processo recursivo.

Ao se remover a célula (3,1,2), é necessário encontrar as vizinhas que ocupem a mesma região no nível 1. Porém, a célula (2,1,2) que será calculada não está presente na malha, então, esta mesma célula é expandida para as suas correspondentes do nível

superior para que também sejam removidas da malha. Esse processo é repetido cada vez que for encontrada uma célula que não está na malha. Quando a ultima célula que está na malha for removida, o processo é encerrado.

As próximas tabelas apresentam o pseudocódigo dos métodos envolvidos na remoção de células da malha.

Quadro 23. Pseudocódigo do método removerCelula

1	**função** removerCelula(índiceX, índiceY, índiceL)
2	**se** tabela.contemCelula(índiceX, índiceY, índiceL) **então**
3	Célula vizinho1
4	Célula vizinho2
5	Célula vizinho3
6	**se** índiceX % 2 == 0 e índiceY % 2 == 0 **então**
7	vizinho1 = {índiceX, índiceY + 1, índiceL}
8	vizinho2 = {índiceX + 1, índiceY, índiceL}
9	vizinho3 = {índiceX + 1, índiceY + 1, índiceL}
10	**senão se** índiceX % 2 == 0 e y % 2 != 0 **então**
11	vizinho1 = {índiceX, índiceY - 1, índiceL}
12	vizinho2 = {índiceX + 1, índiceY, índiceL}
13	vizinho3 = {índiceX + 1, índiceY - 1, índiceL}
14	**senão se** índiceX % 2 != 0 e y % 2 == 0 **então**
15	vizinho1 = {índiceX, índiceY + 1, índiceL}
16	vizinho2 = {índiceX - 1, índiceY, índiceL}
17	vizinho3 = {índiceX - 1, índiceY + 1, índiceL}
18	**senão se** índiceX % 2 != 0 e y % 2 != 0 **então**
19	vizinho1 = {índiceX - 1, índiceY, índiceL}
20	vizinho2 = {índiceX, índiceY - 1, índiceL}
21	vizinho3 = {índiceX - 1, índiceY - 1, índiceL}
22	**fim se**
23	tabela.remover(índiceX, índiceY, índiceL)
24	removerVizinho(vizinho1)
25	removerVizinho(vizinho2)
26	removerVizinho(vizinho3)
27	Célula novaCélula = retrair({índiceX, índiceY, índiceL, falso})
28	tabela.inserir(novaCélula);
29	**fim se**
30	**fim função**

No Quadro 23 é apresentado o código da função principal que realiza o reconhecimento das células vizinhas existentes na mesma região, e faz as remoções necessárias. A célula de interesse que passada por parâmetro através de seus índices é removida da tabela, e as células vizinhas são removidas por meio da função auxiliar *removerVizinho*, que deve verificar se a célula passada como sendo uma suposta vizinha é uma vizinha de fato para então removê-la da tabela de espalhamento. Após todas as remoções serem feitas, a célula de interesse é retraída e a célula obtida neste processo é inserida na tabela de espalhamento.

As células vizinhas são calculadas dentro da estrutura de verificação que está entre as linhas 6 e 22. As células obtidas nestes cálculos são de fato as células que estão na mesma região que a célula de interesse. Isso é garantido pois as verificações de paridade feitas nos índices indicam em qual direção a célula está posicionada, e quando esta informação é conhecida os cálculos são feitos de forma a obter exatamente as células vizinhas que compõem a mesma região. Lembrando que após todas as células serem obtidas, o método auxiliar *removerVizinho* realiza as demais operações sob as células vizinhas.

Quadro 24. Pseudocódigo do método removerVizinho

1	**função** removerVizinho(célula)
2	**se** tabela.contemCelula(célula) **então**
3	tabela.remover(célula)
4	**senão**
5	removerCelulaProximoNivel(célula)
6	**fim se**
7	**fim função**

A função *removerVizinho* é auxiliar para o processo de remoção de células. É esta função que decide se a célula já pode ser removida ou se é necessário realizar uma remoção em próximo nível. Essa decisão é tomada verificando se a célula existe na tabela de espalhamento. Caso a célula exista, ela é removida. Caso não exista é chamada a outra função auxiliar *removerCelulaProximoNivel* onde é feita a remoção em próximos níveis.

Quadro 25. Pseudocódigo da função removerCelulaProximoNivel

1	**função** removerCelulaProximoNivel(célula)
2	**para** Célula c **em** expandir(célula) **faça**
3	**se** tabela.contemCelula(c) **então**
4	tabela.remover(c)
5	**senão**
6	removerCelulaProximoNivel(c)
7	**fim se**
8	**fim para**
9	**fim função**

A função *removerCelulaProximoNivel* é uma função auxiliar do processo de remoção de células da malha que realiza a remoção de células que se encontram dentro de uma mesma região mas que estão em níveis superiores. Como mostrado no exemplo da Figura 14. Nesta função é necessário expandir a célula que é passada como parâmetro pois no método *removerVizinho* foi constatado que a mesma não existe na tabela, então são obtidas as células do próximo nível para continuar a operação. As células do próximo nível são obtidas com o método expandir e a partir das células contidas na lista retornada pelo método, é verificado se a célula existe na tabela de espalhamento. Se sim, a célula é removida da tabela. Se não, o método *removerCelulaProximoNivel* é chamado novamente passando como parâmetro a célula

não encontrada na tabela. Dessa forma ocorre o processo de remoção recursiva de células.

Referências

CORMEN, Thomas H. et al. Algoritmos: Teoria e Prática – 3ª edição. Rio de Janeiro: Elsevier Editora Ltda, 2012.

SZWARCFITER, Jayme Luiz; MARKENZON Lilian. Estruturas de Dados e Seus Algoritmos – 3ª edição. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora Ltda, 2010.

FORTUNA, Armando de Oliveira. Técnicas Computacionais para Dinâmica dos Fluidos - Conceitos Básicos e Aplicações. São Paulo: Edusp - Editora da Univcrsidade de São Paulo, 2000.

ANEXO B – Código Fonte

Arquivo main.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include "tabela.h"
#include "malha.h"

/**
* Os testes realizados neste arquivo são apenas para apresentar alguns recursos implementados.
**/

// Testes na Tabela de Espalhamento

/**
* Método destinado a exibir os dados na tabela de espalhamento.
**/
void exibir(Tabela tabela) {
  int k = 0;
  for (std::vector<Celula> t: tabela.getTabela()) {
    for (Celula c: t) {
      if (!c.nulo) {
        std::cout << k << " {" << c.x << "," << c.y << "," << c.l << "}" << std::endl;
        k++;
      }
    }
  }
}

/**
* Método para o teste do algoritmo de espalhamento.
**/
void testeInsercao() {
  int nivelMalha = 1;
  Tabela tabela = Tabela(nivelMalha);
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int l = 1;
  for (int k = 0; k < pow(4, l); k++) {
    tabela.inserir({x, y, l});
    if (!tabela.contemCelula({x, y, l})) {
      std::cout << "{" << x << "," << y << "," << l << "} não está na tabela." << std::endl;
      break;
    }
    y++;
    if (y == (int)sqrt(pow(4, l))) {
      y = 0;
      x++;
    }
  }
  exibir(tabela);
}

/**
* Este método realiza o espalhamento de células do nível 0 ao nível 12.
* Este teste foi realizado apenas para fins de se analizar a capacidade da tabela
* e o comportamento obtido com a inserção em todos o níveis até 12.
**/
void testeInsercaoMaxima() {
  int nivelMalha = 12;
  Tabela tabela = Tabela(nivelMalha);
  for (int i = 0; i <= nivelMalha; i++) {
    unsigned int x = 0;
    unsigned int y = 0;
    unsigned int l = i;
    for (int k = 0; k < pow(4, l); k++) {
      tabela.inserir({x, y, l});
      if (!tabela.contemCelula({x, y, l})) {
        std::cout << "{" << x << "," << y << "," << l << "} não está na tabela." << std::endl;
        break;
      }
      y++;
      if (y == (int)sqrt(pow(4, l))) {
        y = 0;
        x++;
      }
    }
  }
  exibir(tabela);
  std::cout << "Total de Células: " << tabela.getTotalElementos() << std::endl;
}

/**
* O experimento de inserção aleatória, realizado para se observar a formação
* de células aleatoriamente na tabela de espalhamento.
**/
void experimentoInsercaoAleatoria() {
  int nCelulas = 3000000;
  int nivelMaximo = 11;
  Tabela tabela = Tabela(nivelMaximo);
  while (tabela.getTotalElementos() < nCelulas) {
    for (int i = 0; i < nCelulas; i++) {
      if (tabela.getTotalElementos() == nCelulas) { // Essa verificação garante que o processo de geração de células irá parar ao atingir o total de células em nCelulas
        break;
      }
      unsigned int l = rand()%nivelMaximo+1;
      int fator = pow(2, l);
      unsigned int x = rand()%fator;
      unsigned int y = rand()%fator;
      if (!tabela.contemCelula({x, y, l})) {
        tabela.inserir({x, y, l});
      }
    }
  }
  exibir(tabela);
  std::cout << "Total de Células: " << tabela.getTotalElementos() << std::endl;
}

// Testes na Malha
```

```cpp
/**
* Este método é voltado para a exibição de células vizinhas nos testes de vizinhança.
**/
void exibirVizinhos(Malha malha, Celula celula) {
  std::vector<Celula> vizinhos = malha.vizinhos(celula);
  std::cout << "{" << celula.x << "," << celula.y << "," << celula.l << "}\n" << std::endl;
  for (Celula c: vizinhos) {
    std::cout << "{" << c.x << "," << c.y << "," << c.l << "}" << std::endl;
  }
}

/**
* Método para se testar a operação de busca de células vizinhas na malha.
**/
void testeVizinhos() {
  Malha malha = Malha(3);
  malha.novaCelula(0,1,1);
  malha.novaCelula(1,1,1);
  malha.novaCelula(0,0,2);
  malha.novaCelula(1,0,2);
  malha.novaCelula(2,0,2);
  malha.novaCelula(3,0,2);
  malha.novaCelula(0,1,2);
  malha.novaCelula(1,1,2);
  malha.novaCelula(3,1,2);
  malha.novaCelula(4,3,3);
  malha.novaCelula(5,3,3);
  malha.novaCelula(4,2,3);
  malha.novaCelula(5,2,3);
  exibirVizinhos(malha, {0,1,1});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {1,1,1});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {0,0,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {1,0,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {2,0,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {3,0,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {0,1,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {1,1,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {3,1,2});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {4,3,3});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {5,3,3});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {4,2,3});
  std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {5,2,3});
}

/**
* Um teste de vizinhança em uma malha uniforme de nível 12.
**/
void testeVizinhos2() {
  Malha malha = Malha(12);
  malha.espalhar(12);
  // exibir(malha.getTabela());
  // exibirVizinhos(malha, {3974,4095,12});
  // std::cout << "--------------" << std::endl;
  exibirVizinhos(malha, {829,4094,12});
  // std::cout << "--------------" << std::endl;
  // exibirVizinhos(malha, {4095,4095,12});
}

/**
* Método voltado para o teste de remoção de uma célula de uma malha não-uniforme.
**/
void testeRemocaoCelulaDaMalha() {
  Malha malha = Malha(3);
  malha.novaCelula(0,1,1);
  malha.novaCelula(1,1,1);
  malha.novaCelula(0,0,2);
  malha.novaCelula(1,0,2);
  malha.novaCelula(2,0,2);
  malha.novaCelula(3,0,2);
  malha.novaCelula(0,1,2);
  malha.novaCelula(1,1,2);
  malha.novaCelula(3,1,2);
  malha.novaCelula(4,3,3);
  malha.novaCelula(5,3,3);
  malha.novaCelula(4,2,3);
  malha.novaCelula(5,2,3);
  // malha.removerCelula(1,1,1);
  // malha.removerCelula(0,0,2);
  // malha.removerCelula(2,0,2);
  malha.removerCelula(3,1,2);
  // malha.removerCelula(4,2,3);
  // malha.removerCelula(4,3,3);
  // malha.removerCelula(5,2,3);
  // malha.removerCelula(4,3,3);
  exibir(malha.getTabela());
}

/**
* Teste de remoção de uma célula de uma malha uniforme.
**/
void testeRemocaoCelulaDaMalha2() {
  Malha malha = Malha(3);
  malha.espalhar(3);
  // malha.removerCelula(4,3,3);
  malha.removerCelula(7,7,3);
  exibir(malha.getTabela());
}
```

```c
// Método main

int main() {
  // testeInsercao();
  // testeInsercaoMaxima();
  experimentoInsercaoAleatoria();
  // testeVizinhos();
  // testeVizinhos2();
  // testeRemocaoCelulaDaMalha();
  // testeRemocaoCelulaDaMalha2();
  return 0;
}
```

Arquivo malha.h

```
#ifndef MALHA_H
#define MALHA_H

#include <vector>
#include <map>
#include <set>
#include <algorithm>
#include "tabela.h"

class Malha {
private:
  //! O nível da malha
  int nivel;

  //! A tabela de espalhamento
  Tabela tabela;

  /**
  * Método expandir
  * @brief O método expandir expande uma célula em 4 (mas não as adiciona na malha)
  * @param celula é a célula atual a ser expandida
  * @return A lista com as células produzidas
  **/
  std::vector<Celula> expandir(Celula celula);

  /**
  * Método retrair
  * @brief O método retrair produz uma célula que está em um nível anterior em relação a uma célula atual
  * @param celula se refere a estrutura de célula que será retraida
  * @return A célula que foi obtida no processo
  **/
  Celula retrair(Celula celula);

  /**
  * Método vizinhosProximoNivel
  * @brief O método vizinhosProximoNivel produz as duas células vizinhas que
  * serão encontradas em determinada direção em um nível a frente do nível da célula atual
  * @param celula é a estrutura de célula que será base para a formação de células do próximo nível
  * @param direcao se trata da direção onde a célula deve ser buscada e formada
  * @return A lista de células formadas
  **/
  std::vector<Celula> vizinhosProximoNivel(Celula celula, int direcao);

  /**
  * Método procuraVizinhosProximoNivel
  * @brief O método procuraVizinhosProximoNivel é responsável por encontrar os
  * vizinhos de uma célula atual em níveis a frente, através de uma busca recursiva
  * @param celula é a célula base para a busca
  * @param vizinhos é a lista de células vizinhas, que são adicionadas conforme são encontradas
  * @param direcao se trata da direção onde os vizinhos devem ser buscados
  **/
  void procuraVizinhosProximoNivel(Celula celula, std::vector<Celula> &vizinhos, int direcao);

  /**
  * Método procuraVizinhosNivelAnteior
  * @brief O método procuraVizinhosNivelAnteior é responsável por encontrar os
  * vizinhos de uma célula atual em níveis aanteriores, através de uma busca recursiva
  * @param celula é a célula base para a busca
  * @param vizinhos é a lista de células vizinhas, que são adicionadas conforme são encontradas
  **/
  void procuraVizinhosNivelAnteior(Celula celula, std::vector<Celula> &vizinhos);

  /**
  * Método inserirVizinho
  * @brief O método inserirVizinho insere na lista de vizinhos uma célula que
  * está sendo verificada como vizinha, caso esta célula não seja uma vizinha,
  * é iniciado o segundo passo de busca por vizinhos
  * @param celula é a célula que será analisada como uma possível vizinha
  * @param direcao se trata da direção onde os vizinhos devem ser buscados
  **/
  void inserirVizinho(Celula celula, std::vector<Celula> &vizinhos, int direcao);

  /**
```

```
  * Método removerCelulaProximoNivel
  * @brief O método removerCelulaProximoNivel é auxiliar do método removerVizinho,
  * e faz a remoção recursiva de células da malha até que nenhuma célula que
  * ocupa a mesma região seja encontrada
  * @param celula é a célula que será removida
  **/
  void removerCelulaProximoNivel(Celula celula);

  /**
  * Método removerVizinho
  * @brief O método removerVizinho remove uma determinada célula da malha,
  * e inicia um processo para remover as células que ocupem a mesma região nos próximos níveis
  * @param celula é a célula que será removida
  **/
  void removerVizinho(Celula celula);

public:
  /**
  * Método Construtor
  * @brief O método construtor inicializa a malha
  * @param nivelMalha define o nível da malha
  **/
  Malha(int nivelMalha);

  /**
  * Método espalhar
  * @brief O método espalhar realiza a inserção de células em um determinado nível
  * @param nivelCelula é o nível das células, o qual as células serão inseridas
  **/
  void espalhar(int nivelCelula);

  /**
  * Método novaCelula
  * @brief O método novaCelula é responsável pela inserção de uma única célula na malha
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  **/
  void novaCelula(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método refinarCelula
  * @brief O método refinarCelula produz 4 novas células a partir de uma célula atual
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  **/
  void refinarCelula(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método removerCelula
  * @brief O método removerCelula remove uma célula da malha
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  **/
  void removerCelula(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método vizinhos
  * @brief O método vizinhos retorna as células vizinhas de uma célula atual
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  * @return Uma lista com as células vizinhas
  **/
  std::vector<Celula> vizinhos(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método vizinhos
  * @brief Esta é uma versão auxiliar do método vizinhos, para ser utilizada com estruturas de células formadas
  * @param celula é uma estrutura de célula
  * @return Uma lista com as células vizinhas
  **/
  std::vector<Celula> vizinhos(Celula celula);
```

```
  /**
  * Método getTabela
  * @brief O método getTabela retorna a classe da tabela de espalhamento
  * @return A tabela de espalhamento
  **/
  Tabela getTabela();

  // void create_unstructured_mesh(double (* f) (double x, double y, double t), double tempo);
};

#endif
```

```cpp
Arquivo malha.cpp

#include "malha.h"

/**
* No construtor da malha o nível da malha é atribuído para se ter conhecimento do
* limite que as operações realizadas sob as células podem alcançar. Por exemplo,
* não devem ser inserçidas células que ultrapassem o nível máximo da malha.
* Também é feita a inicialização da tabela de espalhamento.
**/
Malha::Malha(int nivelMalha) {
  nivel = nivelMalha;
  tabela = Tabela(nivelMalha);
}

/**
* A expansão de células é realizada a partir das coordenadas da célula que se deseja
* expandir. As coordenadas para as novas células são obtidas multiplicando-se as coordenadas
* da célula de interesse por 2, e o nível é incrementado.
* A partir desses novos valores, as novas células são obitdas somando-se as coordenadas
* calculadas por 1 no devido momento.
**/
std::vector<Celula> Malha::expandir(Celula celula) {
  unsigned int x = 2*celula.x;
  unsigned int y = 2*celula.y;
  unsigned int l = celula.l+1;
  Celula c1 = {x, y, l};
  Celula c2 = {x, y+1, l};
  Celula c3 = {x+1, y, l};
  Celula c4 = {x+1, y+1, l};
  return std::vector<Celula>{c1, c2, c3, c4};
}

/**
* A retração de uma célula é feita dividindo-se as coordenadas da célula de interesse
* por 2 caso as coordenadas sejam pares, senão as coordenadas são decrementadas e então
* a divisão é feita. O novo nível é obtido decrementado-se o nível da célula da interesse.
* Após obter-se as novas coordenadas, a nova célula é formada.
**/
Celula Malha::retrair(Celula celula) {
  unsigned int x = ((celula.x % 2 == 0) ? celula.x/2 : (celula.x-1)/2);
  unsigned int y = ((celula.y % 2 == 0) ? celula.y/2 : (celula.y-1)/2);
  unsigned int l = celula.l-1;
  return {x, y, l};
}

/**
* Os vizinhos do próximo nível são formados da mesma forma que as células são expandidas,
* basta multiplicar as coordenadas da célula de interesse por 2, e então incrementa-las
* nos devidos momentos.
* Neste método, são produzidas no máximo duas células, pois é o limite máximo de células
* do próximo nível que colidem com uma outra célula em uma determinada direção.
* Caso a direção que se deseja buscar as células seja a diagonal, será produzida
* apenas uma célula.
**/
std::vector<Celula> Malha::vizinhosProximoNivel(Celula celula, int direcao) {
  unsigned int x = 2*celula.x;
  unsigned int y = 2*celula.y;
  unsigned int l = celula.l+1;
  Celula c1, c2;
  switch (direcao) {
    case 1: // Vizinhos Abaixo
      c1 = {x, y, l, false};
      c2 = {x+1, y+1, l, false};
    break;
    case 2: // Vizinhos da Direita
      c1 = {x+1, y, l, false};
      c2 = {x+1, y+1, l, false};
    break;
    case 3: // Vizinhos da Esquerda
      c1 = {x, y, l, false};
      c2 = {x, y+1, l, false};
    break;
    case 4: // Vizinho Acima
      c1 = {x, y, l, false};
```

```cpp
      c2 = {x+1, y, l, false};
    break;
    case 5: // Vizinho Diagonal Direita Superior
      c1 = {x+1, y+1, l, false};
    break;
    case 6: // Vizinho Diagonal Direita Inferior
      c1 = {x+1, y, l, false};
    break;
    case 7: // Vizinho Diagonal Esquerda Superior
      c1 = {x, y+1, l, false};
    break;
    case 8: // Vizinho Diagonal Esquerda Inferior
      c1 = {x, y, l, false};
    break;
  }
  return std::vector<Celula>{c1, c2};
}

/**
* O método procuraVizinhosProximoNivel faz a busca de células vizinhas de uma célula
* de interesse formando estas células com o auxílio do método vizinhosProximoNivel.
* Caso as células formadas existam na tabela de espalhamento, elas são adicionadas na
* lista de vizinhos. Caso não existam, o método procuraVizinhosProximoNivel é chamado
* novamente, passando como parâmetro a célula que não foi encontrada na tabela de
* espalhamento para ser feita a procura de células um nível acima do nível da célula
* não encontrada.
**/
void Malha::procuraVizinhosProximoNivel(Celula celula, std::vector<Celula> &vizinhos, int direcao) {
  if (celula.l < nivel) {
    std::vector<Celula> celulas = Malha::vizinhosProximoNivel(celula, direcao);
    for (Celula c: celulas) {
      if (tabela.contemCelula(c) && !c.nulo) {
        vizinhos.push_back(c);
      } else {
        if (c.l < nivel && !c.nulo) {
          Malha::procuraVizinhosProximoNivel(c, vizinhos, direcao);
        }
      }
    }
  }
}

/**
* O método procuraVizinhosNivelAnteior procura por uma célula vizinha da célula de interesse
* que exista em algum nível anterior ao seu. Essa busca é feita retraindo a célula
* de interesse até se encontrar a célula vizinha. Caso a célula retraída exista na
* tabela, ela é adicionada na lista de vizinhos, do contrário, é retraída novamente
* até a vizinha ser encontrada.
**/
void Malha::procuraVizinhosNivelAnteior(Celula celula, std::vector<Celula> &vizinhos) {
  if (celula.l > 0) {
    if (tabela.contemCelula(celula) && (std::find(vizinhos.begin(), vizinhos.end(), celula) == vizinhos.end())) {
      vizinhos.push_back(celula);
    } else {
      if (celula.l > 0) {
        Celula c = Malha::retrair(celula);
        Malha::procuraVizinhosNivelAnteior(c, vizinhos);
      }
    }
  }
}

/**
* Este método verifica se a célula obtida no primeiro passo de busca existe na tabela
* de espalhamento. Se sim, é adicionada a lista de vizinhos. Se não, é iniciado o
* segundo passo de busca, que procurará células vizinhas nos níveis anteriores e
* superiores da célula de interesse.
**/
void Malha::inserirVizinho(Celula celula, std::vector<Celula> &vizinhos, int direcao) {
  if (tabela.contemCelula(celula)) {
    vizinhos.push_back(celula);
  } else {
    Malha::procuraVizinhosProximoNivel(celula, vizinhos, direcao);
    Malha::procuraVizinhosNivelAnteior(celula, vizinhos);
  }
```

```
}

/**
* A remoção de células do próximo nível é necessária, pois uma célula que está sendo
* removida pode conter vizinhos de um nível superior e que estão ocupando a mesma
* região em uma malha.
* A remoção é feita passando-se uma célula de interesse. Esta célula é expandida e
* caso as células obtidas existam na tabela, elas são removidas. Se uma dessas Células
* não existir na tabela, o método realiza um processo recursivo até que todas as células
* superiores sejam removidas.
**/
void Malha::removerCelulaProximoNivel(Celula celula) {
  for (Celula c: Malha::expandir(celula)) {
    if (tabela.contemCelula(c)) {
      tabela.remover(c);
    } else {
      Malha::removerCelulaProximoNivel(c);
    }
  }
}

/**
* O método remover vizinho é um método auxiliar para removerCelula. Ele simplismente
* remove uma célula indicada da tabela de espalhamento, se esta existir na tabela.
* Caso não exista, é iniciado o processo para a remoção de células do próximo nível.
**/
void Malha::removerVizinho(Celula celula) {
  if (tabela.contemCelula(celula)) {
    tabela.remover(celula);
  } else {
    Malha::removerCelulaProximoNivel(celula);
  }
}

/**
* O método espalhar gera células de um determinado nível, formando uma malha uniforme.
* O espalhamento ocorre inicializando os índice x e y em 0. Estes índices são incrementados
* seguindo um padrão de formação indicado por 4^nivelCelula. O índice y é incrementado enquanto
* seu valor for menor que a raiz de 4^nivelCelula. Quando estes dois valores forem iguais,
* o índice x é incrementado e o índice y é colocado em 0 novamente.
* Todas as células formadas são adicionadas na tabela de espalhamento.
**/
void Malha::espalhar(int nivelCelula) {
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int l = nivelCelula;
  for (int k = 0; k < pow(4, l); k++) {
    tabela.inserir({x, y, l, false});
    y++;
    if (y == (int)sqrt(pow(4, l))) {
      y = 0;
      x++;
    }
  }
}

/**
* O método inserir recebe os índices de uma célula para inseri-la de forma direta
* na tabela de espalhamento.
**/
void Malha::novaCelula(unsigned int x, unsigned int y, unsigned int l) {
  tabela.inserir({x, y, l, false});
}

/**
* Este método cria 4 células do próximo nível a partir de uma célula de interesse.
* As novas células são formadas com o método auxiliar expandir.
* Quando as novas células são formadas, estas são iseridas na tabela de espalhamento,
* e a célula de interesse é removida.
**/
void Malha::refinarCelula(unsigned int x, unsigned int y, unsigned int l) {
  if (tabela.contemCelula(x, y, l)) {
    std::vector<Celula> novasCelulas = Malha::expandir({x,y,l});
    tabela.remover(x, y, l);
    tabela.inserir(novasCelulas[0].x, novasCelulas[0].y, novasCelulas[0].l);
```

```
    tabela.inserir(novasCelulas[1].x, novasCelulas[1].y, novasCelulas[1].l);
    tabela.inserir(novasCelulas[2].x, novasCelulas[2].y, novasCelulas[2].l);
    tabela.inserir(novasCelulas[3].x, novasCelulas[3].y, novasCelulas[3].l);
  }
}

/**
* O método remover célula remove uma célula da malha, mas seu processo é mais elaborado
* pois se trata de uma operação cujo resultado afeta toda a malha.
* Dada uma célula que será removida, são calculadas as suas células vizinhas para
* que também sejam removidas. Isso é necessário pois se for removida apenas a célula
* de interesse, a malha terá um buraco. Então, as células vizinhas na região da célula de
* interesse também são removidas.
* A célua de interesse é removida da tabela de espalhamento, e é chamado o método
* auxiliar removerVizinho para remover os vizinhos calculados.
* O método removerVizinho pode iniciar um processo recursivo, pois caso alguma das células
* vizinhas não exista na tabela de espalhamento, significa que existem células em níveis
* posteriores ocupando sua região, e estas também deverão ser removidas.
* Após todas as células serem removidas, ainda sim existirá um buraco na tabela de espalhamento,
* então é obtida a célula do nível anterior ao nível célula de interesse através do
* processo de retração, para que esta nova célula ocupe o lugar das células removidas.
**/
void Malha::removerCelula(unsigned int x, unsigned int y, unsigned int l) {
  if (tabela.contemCelula(x, y, l)) {
    Celula vizinho1;
    Celula vizinho2;
    Celula vizinho3;
    if ((x % 2 == 0) && (y % 2 == 0)) {
      vizinho1 = {x, y+1, l};
      vizinho2 = {x+1, y, l};
      vizinho3 = {x+1, y+1, l};
    } else if ((x % 2 == 0) && (y % 2 != 0)) {
      vizinho1 = {x, y-1, l};
      vizinho2 = {x+1, y, l};
      vizinho3 = {x+1, y-1, l};
    } else if ((x % 2 != 0) && (y % 2 == 0)) {
      vizinho1 = {x, y+1, l};
      vizinho2 = {x-1, y, l};
      vizinho3 = {x-1, y+1, l};
    } else if ((x % 2 != 0) && (y % 2 != 0)) {
      vizinho1 = {x-1, y, l};
      vizinho2 = {x, y-1, l};
      vizinho3 = {x-1, y-1, l};
    }
    tabela.remover(x, y, l);
    Malha::removerVizinho(vizinho1);
    Malha::removerVizinho(vizinho2);
    Malha::removerVizinho(vizinho3);
    Celula novaCelula = Malha::retrair({x, y, l, false});
    tabela.inserir(novaCelula);
  }
}

/**
* No método vizinhos são obtidas as células vizinhas de uma célula de interesse.
* Esse processo é feito em dois passos. No primeiro passo, são obtidas as células vizinhas
* que estão no mesmo nível da célula de interesse através de calculos simples.
* Se estas células existirem na tabela de espalhamento, elas são adicionadas a lista
* de vizinhos e o segundo passo não é ativado. Caso alguma destas células não exista na tabela,
* o segundo passo é ativado para se procurar as células vizinhas em níveis anteriores e
* posteriores ao nível da célula de interesse.
* O método auxiliar inserirVizinho fica responsável por verificar se as células
* calculadas no primeiro passo existem na tabela, e se será necessário realizar
* o segundo passo de busca.
**/
std::vector<Celula> Malha::vizinhos(unsigned int x, unsigned int y, unsigned int l) {
  std::vector<Celula> vizinhos;
  int fator = (sqrt(pow(4, l)))-1;
  /* Abaixo são decididas em quais direções será necessário realizar a busca por
  células vizinhas. */
  if (y > 0) { // Vizinhos Abaixo
    Celula c = {x, y-1, l};
    Malha::inserirVizinho(c, vizinhos, 1);
  }
  if (x > 0) { // Vizinhos da Direita
```

```
    Celula c = {x-1, y, l};
    Malha::inserirVizinho(c, vizinhos, 2);
  }
  if (x < fator) { // Vizinhos da Esquerda
    Celula c = {x+1, y, l};
    Malha::inserirVizinho(c, vizinhos, 3);
  }
  if (y < fator) { // Vizinhos Acima
    Celula c = {x, y+1, l};
    Malha::inserirVizinho(c, vizinhos, 4);
  }
  if (x > 0 && y > 0) { // Vizinhos da Diagonal Direita Superior
    Celula c = {x-1, y-1, l};
    Malha::inserirVizinho(c, vizinhos, 5);
  }
  if (x > 0 && y < fator) { // Vizinhos da Diagonal Direita Inferior
    Celula c = {x-1, y+1, l};
    Malha::inserirVizinho(c, vizinhos, 6);
  }
  if (x < fator && y > 0) { // Vizinhos da Diagonal Esquerda Superior
    Celula c = {x+1, y-1, l};
    Malha::inserirVizinho(c, vizinhos, 7);
  }
  if (x < fator && y < fator) { // Vizinhos da Diagonal Esquerda Inferior
    Celula c = {x+1, y+1, l};
    Malha::inserirVizinho(c, vizinhos, 8);
  }
  return vizinhos;
}

/**
* Apenas uma versão do método vizinhos para ser usado com estruturas formadas.
**/
std::vector<Celula> Malha::vizinhos(Celula celula) {
  return Malha::vizinhos(celula.x, celula.y, celula.l);
}

/**
* É retornada a tabela de espalhamento utilizada para armazenar as células.
**/
Tabela Malha::getTabela() {
  return tabela;
}
```

```
Arquivo tabela.h

#ifndef TABELA_H
#define TABELA_H

#include <vector>
#include <cmath>
#include <stdexcept>
#include "celula.h"

class Tabela {
private:
  //! A tabela de espalhamento física descrita por um vetor de vetores
  std::vector<std::vector<Celula>> tabela;

  //! O nível da malha (deve ser conhecido pela tabela de espalhamento)
  int nivelMalha;

  //! O contador de elementos adicionados na tabela
  int totalElementos{0};

  /**
  * Método inicializarTabela
  * @brief O método inicializarTabela é responsável por inicializar as tabelas de segundo nível
  **/
  void inicializarTabela();

  /**
  * Método calculoChave
  * @brief O método calculoChave realiza o cálculo da chave intermediária de uma célula
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  * @return O valor final da chave intermediária
  **/
  int calculoChave(unsigned int x, unsigned int y, unsigned int l);

public:
  //! Método Construtor Padrão
  Tabela() = default;

  /**
  * Método Construtor
  * @brief O construtor inicializa a tabela
  * @param nivelMalha é o nível da malha que deve ser conhecido pela tabela
  **/
  Tabela(int nivelMalha);

  /**
  * Método inserir
  * @brief O método inserir é responsável pela inserção de células na tabela
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  **/
  void inserir(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método inserir
  * @brief Esta versão do método insere uma célula pronta na tabela
  * @param celula é a estrutura da célula já formada que será inserida
  **/
  void inserir(Celula celula);

  /**
  * Método remover
  * @brief O método remover torna nula uma célula da tabela
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  **/
  void remover(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método remover
```

```
  * @brief Esta versão do método remover faz a remoção a partir de uma célula formada
  * @param celula é a estrutura da célula formada que será removida da tabela
  **/
  void remover(Celula celula);

  /**
  * Método buscaCelula
  * @brief O método buscaCelula retorna uma célula da tabela a partir de seus índices
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  * @return A celula encontrada
  **/
  Celula buscaCelula(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método contemCelula
  * @brief O método contemCelula retorna se uma célula está na tabela
  * @param x é a coordenada x da célula
  * @param y é a coordenada y da célula
  * @param l é o nível da célula
  * @return Se a célula existe na tabela
  **/
  bool contemCelula(unsigned int x, unsigned int y, unsigned int l);

  /**
  * Método contemCelula
  * @brief Esta versão do método contemCelula procura uma célula formada na tabela
  * @param celula é a estrutura da célula formada que será procurada na tabela
  * @return Se a célula existe na tabela
  **/
  bool contemCelula(Celula celula);

  /**
  * Método getTotalElementos
  * @brief O método getTotalElementos retorna a quantidade de células na tabela
  * @return O total de células na tabela
  **/
  int getTotalElementos();

  /**
  * Método getTabela()
  * @brief O método getTabela retorna a representação física da tabela de espalhamento
  * @return A lista de listas que representa a tabela
  **/
  std::vector<std::vector<Celula>> getTabela();
};

#endif
```

Arquivo tabela.cpp

```
#include "tabela.h"

/**
* Para os métodos que fazem acesso as tabelas de primeiro e segundo nível, são
* calculados os índices para acessar as células de interesse.
* Para se acessar a tabela de primeiro nível, é utilizado o parâmetro l, que representa
* o nível da célula.
* Para acessar a tabela de segundo nível, é feito o cálculo do índice com o método
* de espalhamento da divisão.
**/

/**
* No construtor o nível da malha é atribuído para se determinar o limite maximo
* de inserção de células na tabela.
* A lista de listas que representa a tabela de espalhamento física, é inicializada
* tendo com tamanho o nivel da malha + 1.
* E então é feita a inicialização da tabelas do segundo nível, para se ter os
* espaços de inserção de células devidamente alocados.
**/
Tabela::Tabela(int nivelMalha) {
  this->nivelMalha = nivelMalha;
  tabela = std::vector<std::vector<Celula>>(nivelMalha+1);
  Tabela::inicializarTabela();
}

/**
* Esta método realiza a inicialização das tabelas de segundo nível, onde as tabelas
* são inicializadas entre [0,nivelMalha], tendo como tamanho 4^i, sendo i a variável
* que representa o índice da tabela no primeiro nível.
* O tamanho da tabela de segundo nível é 4^i para se ter o espaço total das Células
* que podem ser inseridas no nível de malha i.
**/
void Tabela::inicializarTabela() {
  for (int i = 0; i <= nivelMalha; i++) {
    tabela[i] = std::vector<Celula>(pow(4, i));
  }
}

/**
* O método calculoChave faz o cálculo da chave intermediária utilizando os parâmetros
* x, y e l que são os índices das células, e a partir do nível l, é calculado
* a quantidade de colunas existentes naquele nível de malha.
* O número de colunas no nível é utilizado para o cálculo final da chave, onde
* o número de colunas é multiplicado com a coordenada y da célula, e depois este
* valor é somado com a coordenada x.
**/
int Tabela::calculoChave(unsigned int x, unsigned int y, unsigned int l) {
  int colunasNivel = pow(2, l);
  int chave = (colunasNivel * y) + x;
  return chave;
}

/**
* Dados os parâmetros da célula, o método de inserção primeiramente
* realiza o cálculo do índice onde a célula será inserida na tabela de segundo
* nível, e utiliza o parâmetro l para indexar a célula na tabela de primeiro nível.
* Caso a célula existente nos índices calculados não seja nula, significa que
* a posição para inserir a célula já está em uso, portanto é reportado um erro em
* tempo de execução indicando uma colisão entre os dados.
* Caso a célula existente nos índices calculados seja nula, a nova célula é inserida
* nas posições indicadas, e o contador da quantidade de elementos na tabela é incrementado.
**/
void Tabela::inserir(unsigned int x, unsigned int y, unsigned int l) {
  int h = Tabela::calculoChave(x, y, l) % tabela[l].size();
  if (!tabela[l][h].nulo) {
    throw(
      std::runtime_error(
        "Colisão: ["+std::to_string(l)+"]["+std::to_string(h)+"] = {"
        +std::to_string(tabela[l][h].x)+","+std::to_string(tabela[l][h].y)+","
        +std::to_string(tabela[l][h].l)+"} e {"+std::to_string(x)+","
        +std::to_string(y)+","+std::to_string(l)+"}"
      )
    );
```

```cpp
  }
  tabela[l][h] = {x, y, l, false};
  totalElementos++;
}

/**
* Este método de inserção tem por objetivo ser aplicado em situações onde a célula
* a ser inserida já está formada, ao ínves de se passar os três parâmetros de índice
* calculados separadamente, é possível realizar a inserção da estrutura da célula
* já concluída.
**/
void Tabela::inserir(Celula celula) {
  Tabela::inserir(celula.x, celula.y, celula.l);
}

/**
* A operação de remoção tem por objetivo anular uma célula existente na tabela
* de espalhamento, para isso, são passados os parâmetros de uma célula, onde
* é calculado o índice da tabela de segundo nível, então é acessada a posição onde
* a célula se encontra nas tabelas de primeiro e segundo nível, onde é atribuida
* uma nova célula com as cordenadas e o nível zerados, e a variavel que indica
* se a célula está nula é setada em verdadeiro.
* O nível da célula também é o indexador da tabela de primeiro nível da célula.
**/
void Tabela::remover(unsigned int x, unsigned int y, unsigned int l) {
  int h = Tabela::calculoChave(x, y, l) % tabela[l].size();
  tabela[l][h] = {0, 0, 0, true};
  totalElementos--;
}

/**
* Este método é um auxiliador para a remoção de uma estrutura de célula já formada.
**/
void Tabela::remover(Celula celula) {
  Tabela::remover(celula.x, celula.y, celula.l);
}

/**
* O método buscaCelula tem por objetivo retornar uma célula a partir de seus parâmetros
* da tabela de espalhamento. Com os parâmetros definidos, é cálculada a posição na tabela
* de segundo nível, para se acessar a célula juntamente com o seu nível na tabela de
* primeiro nível. Se para os parâmetros passados, forem calculadas posições onde
* não existia uma célula inserida, a célula retornada pelo método será uma célula
* nula {0,0,0}.
**/
Celula Tabela::buscaCelula(unsigned int x, unsigned int y, unsigned int l) {
  int h = Tabela::calculoChave(x, y, l) % tabela[l].size();
  return tabela[l][h];
}

/**
* O método contemCelula verifica se uma célula está inserida na tabela de espalhamento.
* Com os parâmetros da célula definidos, é calculado o índice da célula na tabela de
* segundo nível, e então é verificado se os parâmetros da célula existente nas posições
* calculadas são iguais aos parâmetros da célula, se sim, a célula está na tabela.
* Lembrando que o nível l acessa a tabela no primeiro nível.
**/
bool Tabela::contemCelula(unsigned int x, unsigned int y, unsigned int l) {
  int h = Tabela::calculoChave(x, y, l);
  return ((tabela[l][h].x == x) && (tabela[l][h].y == y) && (tabela[l][h].l == l));
}

/**
* Este método checa se uma célula está inclusa na tabela de espalhamento, a partir
* da estrutura da célula já formada.
**/
bool Tabela::contemCelula(Celula celula) {
  return Tabela::contemCelula(celula.x, celula.y, celula.l);
}

/**
* Retorna o total de elementos inseridos na tabela.
**/
int Tabela::getTotalElementos() {
  return totalElementos;
```

```
}

/**
* Retorna a tabela de espalhamento física.
**/
std::vector<std::vector<Celula>> Tabela::getTabela() {
  return tabela;
}
```

Arquivo celula.h

```
#ifndef CELULA_H
#define CELULA_H

/**
* A estrutura que representa a Célula
* x e y são as coordenadas no plano cartesiano 2D
* l é o nível da célula na malha
**/
typedef struct Celula {
  unsigned int x{0};
  unsigned int y{0};
  unsigned int l{0};
  bool nulo{true};
} Celula;

/**
* Método operator==
* @brief O método operator== retorna se duas células possuem parâmetros de índice iguais
* @param c1 é uma estrutura de célula
* @param c2 é uma estrutura de célula
* @return true se os parâmetros x, y e l são iguais em ambas as células,
* do contrário retorna false
**/
inline bool operator==(const Celula c1, const Celula c2) {
  if ((c1.x == c2.x) && (c1.y == c2.y) && (c1.l == c2.l)) {
    return true;
  }
  return false;
}

/**
* Método operator>
* @brief O método operator> retorna se a célua c1 tem ao menos um parâmetro de maior
* valor que c2, porém ambas as células devem estar no mesmo nível
* @param c1 é uma estrutura de célula
* @param c2 é uma estrutura de célula
* @return true se um dos parâmetros x ou y for maior em c1, do contrário retorna false
**/
inline bool operator>(const Celula c1, const Celula c2) {
  if (c1.l == c2.l) {
    if (c1.x == c2.x) {
      if (c1.y > c2.y) {
        return true;
      }
    } else {
      if (c1.x > c2.x) {
        return true;
      }
    }
  }
  return false;
}

/**
* Método operator<
* @brief O método operator< retorna se a célua c1 tem ao menos um parâmetro de menor
* valor que c2, porém ambas as células devem estar no mesmo nível
* @param c1 é uma estrutura de célula
* @param c2 é uma estrutura de célula
* @return true se um dos parâmetros x ou y for menor em c1, do contrário retorna false
**/
inline bool operator<(const Celula c1, const Celula c2) {
  if (c1.l == c2.l) {
    if (c1.x == c2.x) {
      if (c1.y < c2.y) {
        return true;
      }
    } else {
      if (c1.x < c2.x) {
        return true;
      }
    }
  }
```

```
  return false;
}

#endif
```