MIRELLA ANTUNES DE MAGALHÃES

ALGORITMO EFICIENTE PARA CÁLCULO DE MAPAS DE VISIBILIDADE EM TERRENOS ARMAZENADOS EM MEMÓRIA EXTERNA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2009

MIRELLA ANTUNES DE MAGALHÃES

ALGORITMO EFICIENTE PARA CÁLCULO DE MAPAS DE VISIBILIDADE EM TERRENOS ARMAZENADOS EM MEMÓRIA EXTERNA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Aprovada em 24 de abril de 2009

Vladimir Oliveira Di Iorio (Co-Orientador)	Carlos A. Alvares Soares Ribeiro (Co-Orientador)
André Gustavo dos Santos	Clodoveu Augusto Davis Junior

Marcus Vinícius Alvim Andrade
(Orientador)

Agradecimentos

"Nothing we achieve in this world is achieved alone. It is always achieved with others teaching us along the way."
Lee J. Colan

Tenho muito orgulho de escrever esses agradecimentos. Orgulho esse que advém da minha crença de que sozinhos não chegamos muito longe, e poder agradecer me faz lembrar que essa é mais uma vitória que alcanço ao lado daqueles que amo.

Começo agradecendo a Deus por me prover direção todos os momentos da minha vida. Porém o que mais devo a Ele é ter me proporcionado conviver com pessoas maravilhosas para as quais continuo esses agradecimentos.

Agradeço aos meus pais, Jeder e Mariana, por serem meu alicerce e terem me dado a oportunidade dessa conquista me ensinando o valor e a gratificação do conhecimento. Agradeço também à toda a minha família - irmãos, mães (Soninha e Edilza), avós, tios, primos e amigos - que sempre me apoiaram, me enviaram boas energias e comemoraram, de perto ou de longe, minhas vitórias.

Tenho muito a agradecer ao meu marido, Isaac. Nunca poderei esquecer de seu incentivo inicial, quando ainda éramos amigos, seus conselhos me ajudaram a estar onde estou. Também não posso esquecer de seu carinho e compreensão durante todo o tempo dedicado ao mestrado, principalmente durante as noites, madrugadas e fins de semana dedicados ao estudo.

Agradeço também aos amigos e companheiros de trabalho da ATAN por terem entendido a importância e grandiosidade desse momento.

Faço um agradecimento especial ao meu companheiro nesse projeto, Salles, sem ele certamente este trabalho não teria tido tamanha proporção.

Agradeço também especialmente ao meu orientador Prof. Marcus. Não poderia deixar de demonstrar minha gratidão a esse mestre que me acompanhou durante a graduação e acreditou mim para a realização do mestrado, mesmo eu não dedicando tempo integral à leitura de livros e artigos, ao desenvolvimento do trabalho e à escrita dessa dissertação. Com seu estilo de ensinar e pesquisar, ele me apontou a direção certa. A ele meu muito obrigada!

Enfim, agradeço à uma recente e maravilhosa realização: meu bebê. A presença dessa pedacinho de vida em meu ventre me deu energia e força para a conclusão deste trabalho. Nesse momento, mais do que nunca, posso dizer que não obtive essa conquista sozinha!

Índice

Índice de Tabelas

Índice de Figuras

Índice de Códigos

Resumo

MAGALHÃES, Mirella Antunes de, M.Sc., Universidade Federal de Viçosa, Abril de 2009. **Algoritmo eficiente para cálculo de mapas de visibilidade em terrenos armazenados em memória externa**. Orientador: Marcus Vinícius Alvim Andrade. Co-Orientadores: Vladimir Oliveira Di Iorio e Carlos A. Alvares Soares Ribeiro.

Com a maior disponibilidade de dados detalhados de terrenos, muitas aplicações precisam processar grandes áreas geográficas em alta resolução. O processamento massivo de dados envolvido em tais aplicações criou grandes desafios para os SIGs e necessita de algoritmos otimizados tanto para processamento interno quanto para transferência de dados. Uma dessas aplicações é o cálculo de mapas de visibilidade ou *viewshed*, que consiste em obter o conjunto de pontos visíveis a partir de um ponto p. Nesse trabalho, nós apresentamos um estudo e como resultado um algoritmo eficiente para calcular o *viewshed* em terrenos armazenados em memória externa. A complexidade do algoritmo é uma função do número de operações de entrada e saída gastas para calcular a visibilidade e, como mostram os resultados, o algoritmo proposto possui desempenho melhor que os algoritmos conhecidos descritos em literatura.

Abstract

MAGALHÃES, Mirella Antunes de, M.Sc., Universidade Federal de Viçosa, April of 2009. **Efficient algorithm to determine the viewshed in terrains stored in external memory**. Adviser: Marcus Vinícius Alvim Andrade. Co-Advisers: Vladimir Oliveira Di Iorio and Carlos A. Alvares Soares Ribeiro.

As more detailed terrain data become available, many terrain applications must process large geographic areas at higher resolutions. The massive data processing involved in such applications presents significant challenges to GIS and requires algorithms that are optimized for both data transfer and for computation. One of these applications is the viewshed computation, that is, to determine all visible points from a given point p. In this work, we present an study and, as a result, an efficient algorithm to compute the viewshed on terrains stored in external memory. The algorithm complexity is a function of the number of I/O operations required to calculate the viewshed and, as shown by the results, our algorithm is more efficient than other algorithms described in the literature.

Introdução Geral

Nos últimos anos, dados sobre a superfície da Terra têm sido produzidos em larga escala, devido aos avanços tecnológicos na coleta de dados de terrenos, como os sistemas de varredura a radar (LIDAR e IFSAR) (Jense, 2007). Esses sistemas são capazes de gerar rapidamente modelos digitais densos e acurados da topografia, por isso têm sido extremamente utilizados para aplicações que necessitam de maior informação nos modelos digitais de terreno, tais como os Sistemas de Informação Geográfica (SIGs).

Grande quantidade de informações também tem sido criada a partir de várias aplicações como bancos de dados, computação científica, gráficos, entretenimento, multimídia, sensores, aplicações da internet e e-mail. O projeto da NASA *Earth Observing System* (NASA's Earth Observing System (EOS), 2009), parte principal do projeto *Earth Science Enterprise*, produz petabytes (10^{15} bytes) de dados raster por ano. Para se ter idéia da quantidade de dados a que isso se refere, um petabyte equivale aproximadamente ao mesmo volume de informação em um bilhão de livros formatados graficamente. As figuras 1 e 2 representam apenas $6MB$ dentre os petabytes em imagens armazenados pelo EOS no ano de 2008.

Os bancos de dados de imagens de satélites online usados pelo Microsoft TerraServer (parte do MSN Virtual Earth) (TerraServer-USA: Microsoft's online database of satellite images, 2009) e Google Earth (Google Earth online database of satellite images, 2009) possuem o tamanho de vários terabytes (10^{12} bytes). O *data warehouse* do Wal-Mart contém metade de um petabyte (500 terabytes) de dados. Com tal volume de informações, o maior desafio é encontrar formas eficientes de manipulação e processamento, ou grande parte dessa informação se tornará inútil.

Atualmente, algumas das mais importantes aplicações em larga escala envolvem o processamento de dados de terrenos. Aplicações tais como a computação da persistência topológica e a construção da árvore de contorno de terrenos (Aggarwal et al., 2006), o cálculo do fluxo hidrológico e a determinação de bacias hidrográficas (Arge et al., 2003) são alguns exemplos. O cálculo dos pontos visíveis de um terreno

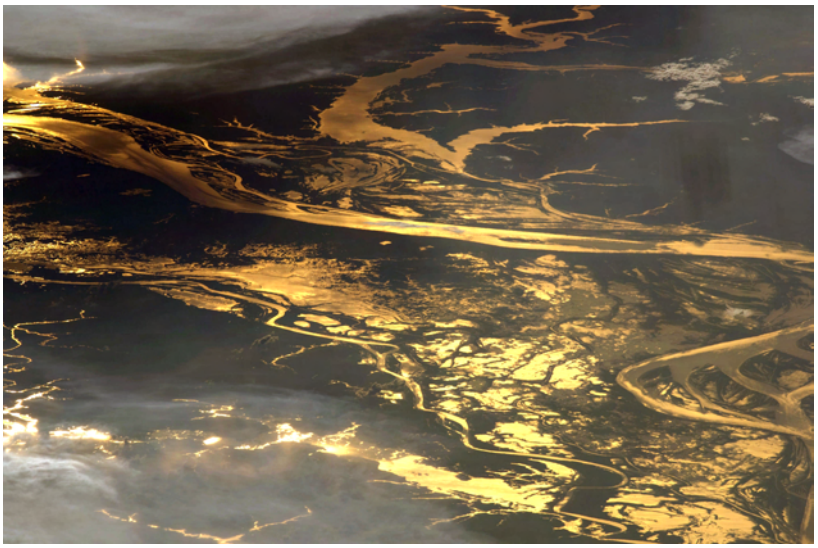


Figura 1: Imagem do Rio Amazonas tirada por astronauta em agosto de 2008.

Figura 2: Imagem da Cratera da Serra da Cangalha localizada no Tocantins.

a partir de um observador, frequentemente chamado de mapa de visibilidade ou *viewshed*, também é uma dessas aplicações. Esse cálculo é utilizado no posicionamento de múltiplos observadores que consiste em encontrar o número mínimo de "observadores" necessários para "cobrir visualmente" uma área específica do terreno, ou então encontrar a área máxima coberta por um dado número de "observadores". Através do posicionamento de múltiplos observadores, o cálculo do *viewshed* possui diversas e

importantes utilizações práticas, tais como determinar o número mínimo de torres de celular necessárias para cobrir uma região (Ben-Moshe et al., 2007; Bespamyatnikh et al., 2001; Murray et al., 2007; Rana, 2005), otimizar o número e a posição de guardas para cobrir uma região (Franklin e Vogt, 2006; Eidenbenz, 2002; Caldwell et al., 2003), otimizar o planejamento de rotas em terrenos (Lee e Stucky, 1998), etc.

Além da importância das aplicações envolvendo o cálculo do *viewshed*, a maior motivação desse trabalho é o aumento significativo da quantidade de computação necessária para a execução dessa aplicação em terrenos maiores que a memória interna. A Tabela 1 apresenta o tempo de execução para determinar o *viewshed* em terrenos de diferentes tamanhos pelo algoritmo de Van Kreveld (Van Kreveld, 1996), que é um dos algoritmos mais eficientes para o cálculo do *viewshed* em memória interna. Vale ressaltar que esses dados foram obtidos no trabalho (Haverkort et al., 2007) e foram coletados utilizando uma máquina Power Macintosh G5 com processador dual de 2.5GHz, 512 KB de cache por processador, 1GB de RAM e HD serial-ATA de 7200 RPM. De acordo com a Tabela 1, um aumento modesto no tamanho dos modelos de terreno causa uma explosão de custo inaceitável, gerando até mesmo a impossibilidade de processamento dos terrenos.

Tamanho do Terreno	Tempo de execução (s)
7 MB	8
25 MB	38
40 MB	80
119 MB	736
267 MB	falha de alocação de memória

Tabela 1: Tempo de execução do algoritmo de Van Kreveld

Embora uma grande quantidade de dados signifique uma maior riqueza de detalhes e aumente o potencial das aplicações, isto causa problemas de escalabilidade para os algoritmos existentes, pois o aumento dos requisitos computacionais necessários para o bom desempenho do algoritmo supera em muito o aumento dos dados a serem processados. Por exemplo, o tamanho da memória interna tem sofrido aumento devido às novas tecnologias, mas não com a rapidez necessária para satisfazer a necessidade de aplicações em larga escala que, para alcançar a eficiência, precisam de todos os dados acessíveis a partir da memória interna.

Alguns dos algoritmos para cálculo de *viewshed* de terrenos tentam reduzir o tamanho dos modelos digitais e, consequentemente, o tempo de execução, através de simplificações nos dados dos terrenos, eliminando características que, de acordo com algum critério, podem não ter influência sobre a visibilidade dos pontos. Outra

solução utilizada é a diminuição da resolução da representação do terreno. Porém, essas estratégias podem suprimir os efeitos de variações relativamente pequenas nos dados de elevação e podem gerar um grande impacto na visibilidade, pois essa é uma aplicação onde qualquer diferença do modelo digital para o terreno original pode causar distorções inaceitáveis no resultado. Além disso, geralmente, qualquer tentativa de diminuição de tamanho dos modelos digitais é insuficiente para evitar que uma parte dos dados seja armazenada em memória externa, dado o grande tamanho dos modelos gerados atualmente.

Portanto, a maioria dos algoritmos atuais para cálculo de *viewshed* não são capazes de armazenar e processar os modelos digitais internamente, exigindo que estes dados sejam processados em memória externa. Isso torna esses algoritmos pouco eficientes pois, ao lidar com dados que não cabem na memória interna, o tempo gasto nas operações de entrada e saída resultantes do acesso e transferência de dados de/para a memória externa é muito maior que o tempo para processamento interno. Por isso, os algoritmos que realizam este tipo de processamento precisam minimizar o acesso aos dados.

Mais especificamente, aplicações em larga escala que manipulam dados em memória externa devem ter seus algoritmos projetados e analisados sob um modelo computacional que avalia a complexidade considerando as operações de transferência de dados ao invés das operações de processamento na CPU. Diversos modelos têm sido criados propondo determinar a complexidade dos algoritmos considerando o número de operações de entrada e saída executadas; um deles é o modelo de Aggarwal e Vitter (Aggarwal e Vitter, 1988). Algoritmos que utilizam esse tipo de modelo são geralmente chamados de algoritmos em memória externa ou *I/O efficient algorithms*.

Algoritmos em memória externa têm o objetivo de diminuir o número de acessos aos dados armazenados externamente. Para isso, esses algoritmos utilizam estruturas de dados em disco que organizam e permitem o acesso aos dados de forma eficiente, além de algoritmos que manipulam esses dados na própria memória externa, utilizando o mínimo possível de operações de entrada e saída. Devido a pesquisas extensivamente realizadas sobre a manipulação de grande volume de dados, esses algoritmos podem se concentrar na resolução do problema a que se propõem com o auxílio de bibliotecas que disponibilizam diversas estruturas e algoritmos que processam dados em memória externa, encapsulando a forma como esse processamento é realizado (Arge et al., 2002a; Dementiev et al., 2005).

O objetivo dessa dissertação é apresentar um algoritmo para cálculo de *viewshed* em terrenos armazenados em memória externa, o EMVS (*External Memory*

Viewshed). A estratégia de desenvolvimento do EMVS consistiu de diversas etapas incluindo a escolha de um modelo de representação para o terreno utilizado pelo algoritmo; o estudo e a definição de um algoritmo em memória interna para ser usado como base para o algoritmo em memória externa [1]; o estudo dos melhores modelos e técnicas de tratamento de dados em memória externa; o projeto e desenvolvimento do algoritmo em si; e estudos comparativos com outros algoritmos conhecidos que resolvam o mesmo problema para atestar a eficiência do EMVS.

Todas as etapas de desenvolvimento do EMVS foram divididas em capítulos e serão apresentadas ao longo da dissertação. Mas, resumidamente, o EMVS é um algoritmo eficiente para cálculo do *viewshed* de pontos em terrenos representados por MDEs [2] armazenados em memória externa. O algoritmo é uma adaptação do método desenvolvido por Franklin e Ray (Franklin, 2002) para cálculo de *viewshed* em memória interna permitindo a manipulação eficiente de grandes terrenos (6GB ou mais). O grande número de acessos ao disco é otimizado pelo uso da biblioteca STXXL (Dementiev et al., 2005). A complexidade do EMVS é analisada sob o modelo proposto por Aggarwal e Vitter (Aggarwal e Vitter, 1988). Comparando nosso algoritmo com o algoritmo de Franklin e Ray, o nosso se mostrou 3.5 vezes mais rápido e conseguiu processar terrenos maiores que 5 GB enquanto que o algoritmo de Franklin e Ray só foi capaz de processar terrenos de, no máximo, 2 GB. Além disso, comparando o EMVS com o algoritmo proposto por Haverkort et al. (Toma et al., 2007), também proposto para calcular o *viewshed* em memória externa, podemos dizer que o EMVS se mostrou mais de 7 vezes mais rápido, além de ser mais simples e fácil de implementar.

Organização da Dissertação

O Capítulo 2 descreve fundamentos teóricos que serviram de base para o projeto do algoritmo em memória externa. Na Seção 2.1, apresentamos os modelos de representação de terrenos mais conhecidos e os critérios utilizados para a definição do modelo a ser utilizado no projeto do algoritmo EMVS. A Seção 2.2 apresenta detalhes dos diversos problemas de visibilidade em terrenos, incluindo a determinação do *viewshed*. A Seção 2.3 apresenta o resultado de pesquisas recentes sobre a manipulação de grandes volumes de informações incluindo a definição de modelos computacionais e estruturas de dados para projeto e análise de algoritmos para manipulação de dados em memória externa.

[1] A idéia foi adaptar um algoritmo já existente para acessar e processar de forma eficiente os dados em memória externa

[2] Modelos Digitais de Elevação: matrizes que armazenam a elevação dos pontos do terreno

O Capítulo 3 é dedicado aos trabalhos relacionados ao assunto da dissertação: visibilidade em terrenos e algoritmos em memória externa. Na Seção 3.1, apresentamos os diversos algoritmos para cálculo de *viewshed* em MDE's enquanto na Seção 3.2 descrevemos algoritmos recentes que utilizam técnicas de processamento de dados em memória externa.

A principal função do Capítulo 4 é descrever o algoritmo EMVS. Para tanto, a Seção 4.1 apresenta o algoritmo de Franklin e Ray (Franklin, 2002), usado como base para o EMVS e a Seção 4.2 descreve o EMVS apresentando detalhes de sua implementação. A Seção 4.3 apresenta a análise de complexidade do algoritmo.

O Capítulo 5 mostra a análise de eficiência do EMVS apresentando os resultados das comparações realizadas para atestar seu desempenho. Nessa análise, utilizamos duas estratégias: comparar o algoritmo em memória externa com o algoritmo de Franklin e Ray (Franklin, 2002) para mostrar sua escalabilidade e compará-lo com um outro algoritmo em memória externa para cálculo do *viewshed*, o algoritmo de Haverkort et al. (Toma et al., 2007), para mostrar seu ganho de eficiência. A Seção 5.1 mostra a descrição do algoritmo de Haverkort et al. A Seção 5.2 mostra a análise realizada para comprovar a eficiência do algoritmo EMVS. Já as seções 5.3 e 5.4 apresentam a comparação com o algoritmo de Haverkort et al. e com o algoritmo de Franklin e Ray, respectivamente.

Enfim, no último capítulo, apresentamos a conclusão do trabalho e enumeramos trabalhos futuros.

Capítulo 1

Referencial Teórico

Para o projeto do algoritmo apresentado nesta dissertação, precisamos definir um modelo de representação de terrenos, entender como ocorre o cálculo da visibilidade em terrenos e escolher as melhores técnicas de manipulação de dados em memória externa. Todos os conceitos sobre essas questões serão apresentados nas próximas seções.

1.1 Representação de superfícies de terreno

Do ponto de vista matemático, um terreno (superfície topográfica) T pode ser visto como a imagem de uma função bidimensional f definida sobre um domínio D compacto e conectado no plano Euclidiano (De Floriani e Magillo, 2003; De Floriani et al., 1999), isto é, $T = (x, y, f(x, y)), x, y \in D$. Um modelo digital de terreno (MDT) é um modelo em que a superfície do terreno é construída a partir de um conjunto finito de dados digitais (medidas de elevação) $V = v_0, ..., v_n \in D$ e, possivelmente, um conjunto de segmentos de linha não-sobrepostos $E = e_0, ..., e_m$ cujas extremidades pertencem a V. Um MDT construído em V representa uma superfície que interpola, ou aproxima, as medidas de elevação dos pontos de V.

Os dados em V representam a variação da altitude ao longo do terreno. Para que esses dados, também chamados de amostras, sejam representativos, deve-se considerar sua quantidade e também o seu posicionamento com relação à distribuição de altimetria na superfície (Felgueiras, 2001). Por exemplo, uma super-amostragem numa região plana significa redundância de informação, enquanto que poucos pontos em uma região de relevo movimentado indicam escassez de informação.

As caracterizações descritas acima apresentam duas etapas do processo de modelagem de um MDT: aquisição das amostras ou amostragem e geração do modelo

propriamente dito ou modelagem. Existe ainda uma terceira etapa definida como a utilização do modelo ou aplicações, veja a Figura 1.1. Cada etapa é decisiva para a qualidade do MDT gerado, sendo também interdependentes, ou seja, uma pode influenciar diretamente na outra. Entretanto, existe uma ordem lógica de estruturação para construção de um MDT, devendo-se inicialmente saber a razão da geração do modelo, ou seja, o seu objetivo e a sua aplicação. A partir dessa definição, já se pode ter noção da área de trabalho, do tipo de terreno que se quer modelar, da disponibilidade de dados de entrada (amostragem) e do tipo de modelo que venha a ter maior eficiência para o terreno representado, tendo em vista um conjunto de aplicações.

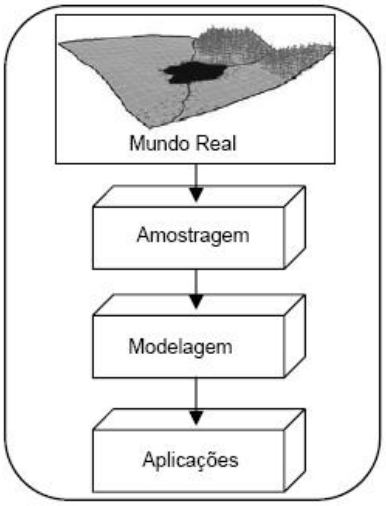


Figura 1.1: Etapas do processo de desenvolvimento de um MDT

O tipo de modelo depende da forma como os pontos de V (amostras) são escolhidos no terreno. Uma escolha aleatória (espalhada) caracteriza a TIN (*Triangulated Irregular Network*), a escolha de pontos distribuídos sobre uma grade regular caracteriza o modelo digital de elevação raster, também chamado de DEM (*Digital Elevation Model*) ou MDE (Modelo Digital de Elevação), e a geração de isolinhas caracteriza os Mapas de Curvas de Nível, veja Figura 1.2.

Uma *TIN* aproxima superfícies por poliedros de faces triangulares não sobrepostas, um exemplo está na Figura 1.2(a). Geralmente os vértices dos poliedros são os pontos amostrados da superfície. Este modelo pode ser visto como uma partição do domínio D em regiões triangulares cujos vértices são pontos de V e cujas bordas são segmentos de E. Neste caso, a função f é definida por partes, como sendo uma função linear sobre cada face triangular.

O modelo de TIN mais popular em softwares comerciais é aquele gerado a partir da Triangulação de Delaunay, um procedimento que usa como critério a maximização

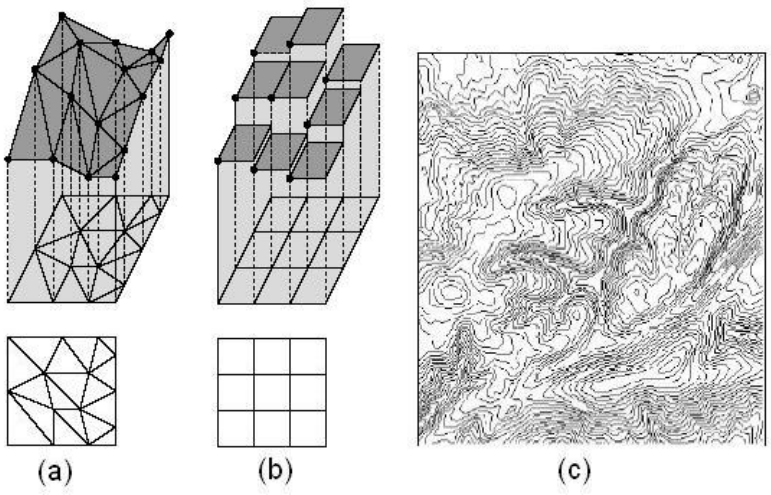


Figura 1.2: Classes de modelos digitais de terreno: (a) TIN; (b) MDE; (c) Mapa de Curvas de Nível

dos ângulos mínimos de cada triângulo (Schewchuk, 1996). Esse critério tem como objetivo evitar a criação de triângulos afinados ou com ângulos internos muito agudos, ou seja, a superfície final deve conter triângulos tão próximos de equiláteros quanto possível.

Matematicamente, um MDE consiste de uma partição do domínio D utilizando uma malha regular de retângulos ou quadrados de mesmo tamanho (geralmente, cada retângulo é denominado pixel em analogia às imagens digitais), veja Figura 1.2(b). Neste caso, a função f (também chamada de função de interpolação) é definida por partes sobre cada retângulo e depende do grau de continuidade desejado. Esse grau de continuidade está relacionado com a forma como serão estimados os valores de elevação de pontos que não fazem parte da matriz, que é composta apenas dos pontos amostrados do conjunto V.

Funções de interpolação são usadas em ambas as representações, MDE e TIN, quando é necessário saber os valores de elevação de pontos que não fazem parte da matriz ou não são os vértices dos triângulos. As funções de interpolação podem ser definidas como globais ou locais. A função global é dependente de todos os pontos amostrados, sendo que a alteração de um valor de entrada afeta todos os pontos do modelo. As funções locais são definidas para pequenas porções dos modelos, sendo aplicadas sucessivamente até cobrir toda a área dos mesmos. Essas porções podem ser definidas por raios de influência, ou por quantidade de amostras vizinhas. Nesse caso, uma alteração num valor de entrada afeta apenas o resultado de um subconjunto.

Em geral o conjunto de amostras pode ser muito grande e não homogêneo,

tornando a interpolação global pouco apropriada tanto em relação ao tempo de processamento quanto à precisão do modelo. Assim, os interpoladores locais são mais utilizados que os globais. Os interpoladores locais mais comuns são os lineares, bilineares e os interpoladores pelo inverso da distância dos vizinhos mais próximos.

A interpolação linear envolve a manipulação de triângulos, portanto, para o MDE é necessário dividir os retângulos em dois triângulos e para a TIN os próprios triângulos do modelo são utilizados. A interpolação linear determina a equação dos planos que passam pelos três vértices de cada um dos triângulos do modelo e as elevações para os pontos situados entre os vértices dos triângulos são calculadas através da equação do plano que cobre esse ponto.

O interpolador bilinear possui esse nome por se tratar de uma interpolação linear tanto na horizontal quanto na vertical. Para um MDE, esse interpolador aproveita as características de ordenação das posições de cada um dos vértices dos retângulos. Por exemplo, na Figura 1.3, a interpolação bilinear sobre o retângulo $ABCD$, com o objetivo de determinar a elevação do ponto M, é realizada em dois passos:

- Interpola-se linearmente os pontos E e F a partir dos pontos D e C, e A e B, respectivamente;
- Interpola-se o ponto M linearmente a partir dos pontos E e F.

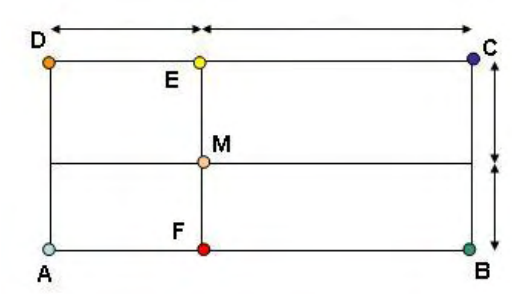


Figura 1.3: Interpolação bilinear

Já o interpolador pelo inverso da distância dos vizinhos mais próximos calcula o valor de cada ponto através de uma média dos valores de seus pontos vizinhos usando o inverso da distância euclidiana entre o ponto e cada um dos seus vizinhos como fator de ponderação (Burrough, 1986).

Na literatura, há diversos trabalhos (Kumler, 1994; Moore et al., 1993; Berg et al., 2000) descrevendo métodos para realizar a conversão entre as três formas principais de representação de terrenos e também apresentando as vantagens e desvantagens

de cada uma delas. Dentre esses trabalhos, (Kumler, 1994) assume o MDE como melhor modelo, e (Moore et al., 1993; Berg et al., 2000), defendem a utilização de TIN. Ainda não existe consenso definitivo sobre qual destas representações é a melhor mas, como descrito nesses trabalhos, na grande maioria das situações, as análises computacionais são realizadas utilizando TIN e/ou MDE, pois embora um Mapa de Curvas de Nível possa ser facilmente interpretado pelas pessoas, esta forma de representação não é muito adequada para o processamento computacional devido à ausência de informações sobre a morfologia do terreno entre duas linhas do mapa.

Comparando TIN e MDE, podemos perceber que a vantagem principal de uma TIN é a possibilidade de uso de resolução variável incluindo-se muitos pontos onde a superfície é acidentada e, ao mesmo tempo, poucos pontos em áreas onde a superfície é relativamente uniforme. Além disso, uma TIN possui a habilidade de manter no modelo pontos importantes da superfície tais como linhas de quebra (drenagem, divisores, falhas, etc). Em contrapartida, a TIN gera grande complexidade na forma de armazenamento e manipulação do modelo. A localização de todo ponto precisa ser especificada nas dimensões x, y e z, enquanto que no modelo MDE a localização horizontal é implícita de acordo com a ordem da matriz. Além disso, na TIN a manipulação dos modelos exige algoritmos complexos para realização de diversos cálculos tais como adjacências de triângulos. Já o MDE proporciona facilidade de geração e manipulação dos dados, utiliza uma estrutura muito simples de armazenamento e é bastante popular devido a sua ampla utilização em sistemas de informações geográficas. Esse modelo requer muito espaço de memória, mas seus algoritmos são bem mais simples e intuitivos. Devido à sua simplicidade, neste trabalho, consideramos terrenos representados por MDE.

1.2 Visibilidade em Terrenos

A maioria dos problemas de SIG relacionados à visibilidade se concentram em determinar a visibilidade de um ponto específico, chamado de observador, e no uso da visibilidade para resolver problemas de otimização. Alguns exemplos de problemas de otimização são o posicionamento ótimo de recursos, a minimização do posicionamento de guardas, planejamento de rotas, etc.

Os problemas de visibilidade podem ser classificados em duas categorias principais: determinação de visibilidade e cálculo de estruturas de visibilidade (De Floriani e Magillo, 2003).

1.2.1 Estruturas de Visibilidade

As principais estruturas de visibilidade calculadas para um observador p são o mapa de visibilidade ou *viewshed* e o horizonte ou *horizon*. O *viewshed* é o conjunto dos pontos de um terreno T visíveis pelo observador, isto é, dado um observador situado no ponto p

$$viewshed(p) = \{q \in T |\ q \text{ é visível por } p\}$$

Algumas vezes, o *viewshed* é restrito a uma região próxima do observador, ou seja, considera-se apenas os pontos a uma distância r do observador. Essa distância é normalmente denominada *raio de interesse* do terreno. Assim, os possíveis pontos visíveis estão restritos a um círculo centrado em p e com raio r, isto é

$$viewshed(p, r) = \{q \in T |\ \text{distancia}(p,q) \leq r \text{ e } q \text{ é visível por } p\}$$

A menos que explicitamente dito o contrário, quando o raio de interesse estiver definido, usaremos $viewshed(p)$ para se referir a $viewshed(p, r)$.

A Figura 1.4 mostra, em (a), o terreno do lago *Champlain West* e, em (b), o seu *viewshed* utilizando o observador no centro do terreno e limitando o raio de interesse. A área escura na Figura (b) representa a porção do terreno, dentro do raio de interesse, vista pelo observador.

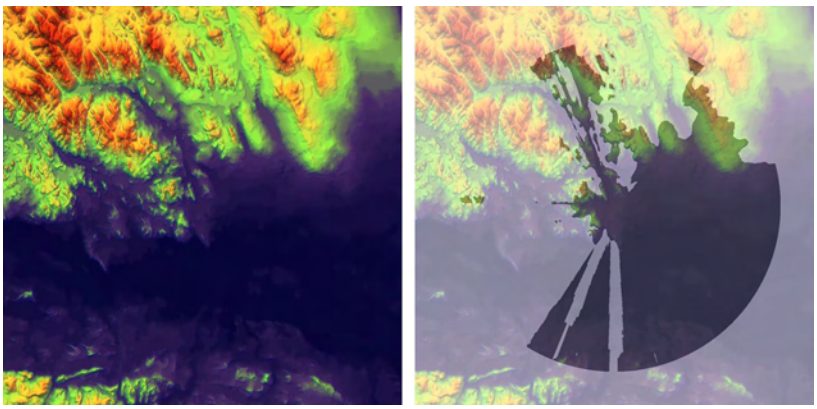


Figura 1.4: Viewshed - Terreno do lago *Champlain West* - Divisa EUA-Canadá

O horizonte pode ser classificado em horizonte local e horizonte global. O horizonte local corresponde à localização dos pontos de um terreno T que são visíveis por p e bloqueiam a visibilidade dos pontos do terreno que estão além deles:

$$localhorizon(p) = \{q \in T |\ q \text{ é visível por } p \text{ e não existe } s \in T, \text{ distinto de } p,$$
$$\text{tal que } q \text{ está em } \overline{ps} \text{ e qualquer ponto em } \overline{qs} \text{ é visível por } p\ \}$$

O horizonte global é frequentemente chamado apenas de horizonte e corresponde ao limite distal do *viewshed*, ou mais precisamente, corresponde ao ponto mais longe do terreno visível a partir de p, para cada direção radial ao redor de p no plano xy:

$$horizon(p) = \{q \in T | q \text{ é visível por } p \text{ e todo } s, \text{ tal que } q \text{ está em } \overline{ps}, \text{ não é visível por } p\}$$

Normalmente, o *viewshed* é representado por uma matriz de lado $2r$. Cada célula armazena 1 ou 0 para indicar se essa célula é visível ou não, respectivamente. Porém alguns autores (Fisher, 1996; Van Kreveld, 1996) propõem uma extensão do *viewshed* de forma a incluir na matriz para cada ponto $q \in T$ mais informações além dessa classificação booleana. Nesse caso, a matriz poderia armazenar informações tais como:

- um valor indicando alguma das seguintes situações: q não é visível por p, q pertence ao horizonte global, q pertence ao horizonte local ou q é visível e não está em nenhum horizonte.
- um *offset* local, definido pela diferença entre a altura de q e o horizonte local mais próximo de q. Esse valor pode ser positivo ou negativo dependendo da visibilidade do ponto, claramente pontos que pertençam a um horizonte local têm valor 0.
- um *offset* global, definido de forma similar ao *offset* local, porém considerando horizonte global.

A vantagem dessa extensão é que a matriz armazena informações suficientes para definir quais pontos pertencem ao *viewshed*, ao horizonte local e ao horizonte global. Veja Figura 1.5.

A partir dos *viewsheds* de um conjunto de observadores no terreno, podemos gerar uma *estrutura de visibilidade* que os engloba. Essa estrutura geralmente é obtida através da aplicação de alguns operadores sobre os *viewsheds* dos observadores em questão. Alguns operadores mais comuns são:

- sobreposição: esse operador sobrepõe os *viewsheds* de todos os observadores e marca cada parte visível do terreno T com o *viewshed* que gerou essa visibilidade.

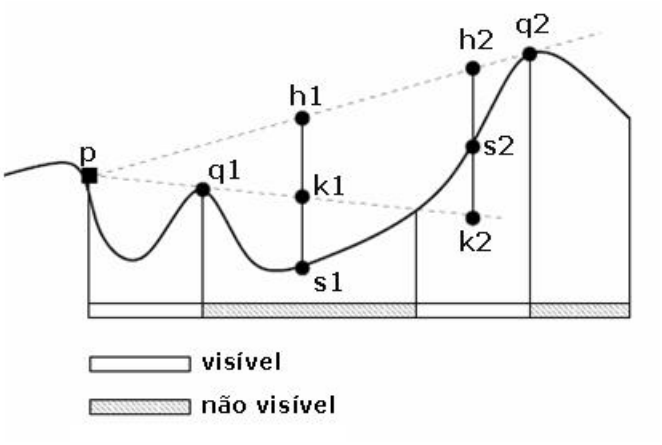


Figura 1.5: q_1 e q_2 são horizontes locais, q_2 é um horizonte global; k_1 é o *offset* local de s_1 e h_1 é seu *offset* global, o mesmo ocorre para s_2; s_1 não é visível por p

- operadores booleanos: usam operações como interseção e união sobre os *viewsheds* dos observadores para determinar a estrutura. Por exemplo, a interseção gera uma estrutura onde os pontos são vistos por todas os *viewsheds*, já a união gera uma estrutura cujos pontos são vistos por pelo menos um *viewshed*;
- operadores de contagem: é similar ao operador de sobreposição, porém nesse caso cada parte visível do terreno T é marcada com o número de *viewsheds* que a "enxergam". Nesse caso, não é possível distinguir quais *viewsheds* "enxergam" cada parte;

Os *índices de visibilidade* são representações discretas das estruturas de visibilidade obtidas através do operador de contagem. Nessa representação, cada ponto do terreno T é marcado com o número de observadores que o veem. O índice de visibilidade também pode ser usado de forma singular para um ponto p, nesse caso, o índice é denotado por $vix(p)$.

1.2.2 Determinação de Visibilidade

A determinação de visibilidade consiste em verificar se um dado ponto é visível ou não a partir de um outro ponto situado no terreno. Isso pode ser determinado assumindo que um ponto q é visível a partir de outro ponto p se, e somente se, o segmento de reta $\overline{pq}$, denominado linha de visada, ou LOS (*line of sight*), está estritamente acima do terreno (exceto nos pontos p e q). Veja a Figura 1.6.

Em uma solução força-bruta, esse problema se reduz a encontrar as arestas do terreno (para uma TIN), ou as células da matriz (para um MDE) interceptadas pelo

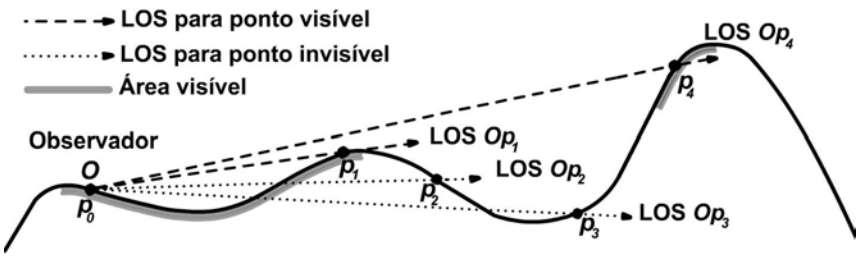


Figura 1.6: Visibilidade de pontos: p_1 e p_4 são visíveis a partir de p_0; p_2 e p_3 não são visíveis a partir de p_0.

plano vertical que passa pelo segmento $\overline{pq}$. Para cada elemento interceptado (aresta ou célula) e, um teste é realizado para decidir se e fica acima de $\overline{pq}$, e q é dito não visível por p no caso de haver uma resposta positiva para pelo menos um desses testes.

Em um MDE, os pontos do terreno são associados às células da matriz a que eles pertencem, enquanto as linhas são obtidas através de agregações de células. Para a determinação de visibilidade entre os dois pontos p e q nesse modelo, é necessário determinar inicialmente quais células pertencem ao segmento $\overline{pq}$ e que, portanto, podem interferir na visibilidade dos pontos. Esse processo é conhecido como rasterização de linhas. Veja Figura 1.7.

Em um algoritmo imediato para rasterização de linhas, os pontos pertencentes ao segmento $\overline{pq}$ são identificados através da equação linear

$$y = mx + b$$

Supondo que os pontos p e q possuem as coordenadas (x_1, y_1) e (x_2, y_2) respectivamente, então os coeficientes m e b da equação acima são

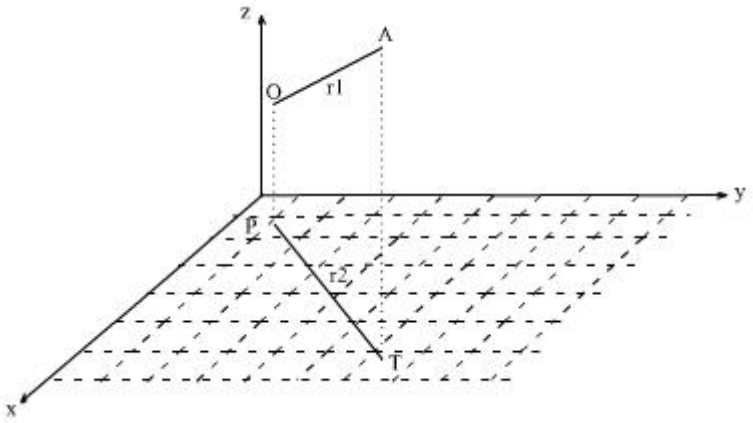
$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$b = y_1 - mx_1$$

Assim, fazendo x variar entre x_1 e x_2 por incrementos de uma unidade, o valor de y é o valor inteiro mais próximo obtido pelas expressões acima. Isto é dado por

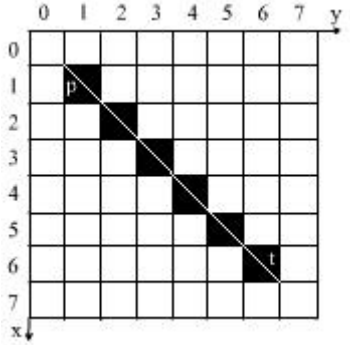
$$y = Round(mx + b) = Floor(0,5 + mx + b)$$

em que *Round* é a função de arredondamento matemático e *Floor* é a função de truncamento.

Após a determinação dos pontos do MDE pertencentes ao segmento $\overline{pq}$, é necessário determinar se cada um desses pontos impede a visibilidade entre p e q. Um ponto p' pertencente a $\overline{pq}$ impede a visibilidade de q se a altura h' do ponto p' no terreno é maior que a altura h do ponto q no terreno (De Floriani et al., 1999).



(a)



(b)

Figura 1.7: Rasterização de linha: células mais escuras são a representação *raster* da linha que passa por elas

Para uma TIN, o problema de determinar se q é visível por p pode ser reduzido a uma instância especial do problema de *ray shooting* em um terreno poliédrico (De Floriani e Magillo, 2003). Dado um terreno poliédrico τ, um observador p e um sistema de coordenadas esféricas centradas em p, um raio de visão r é um raio que emana do ponto p e é identificado pelo par (θ, α), chamado de direção de visão, onde θ é o ângulo entre a projeção $\overline{r}$ de r no plano xy no eixo positivo de x, e α é o ângulo entre r e o eixo x, veja Figura 1.8. Nesse caso, o problema de *ray shooting* consiste em determinar a primeira face f de τ a ser atingida pelo raio r que emana de p e passa por q. O ponto q é visível se, e somente se, a face f não intercepta o segmento $\overline{pq}$. Veja Figura 1.9.

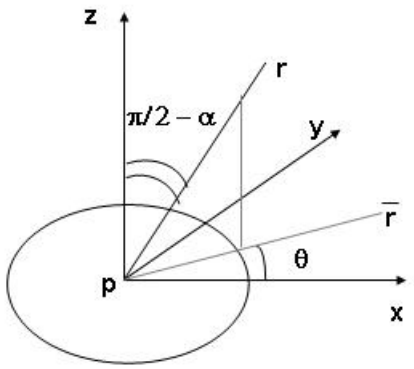


Figura 1.8: Sistema de coordenadas esféricas

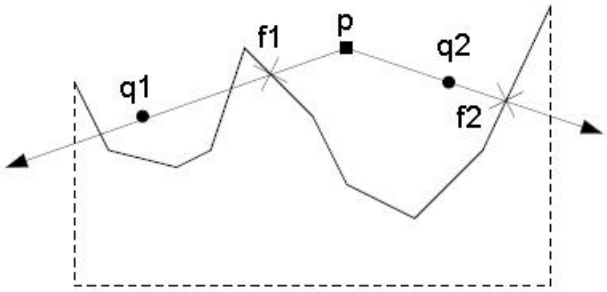


Figura 1.9: Visibilidade em TIN: ponto q_1 não é visível por p devido a f_1 e o ponto q_2 é visível por p.

1.3 Algoritmos eficientes em memória externa

Em aplicações em larga escala, que processam grande volume de informações, as operações de entrada e saída (E/S) entre a memória interna e a memória externa (como os discos) podem se tornar o gargalo da computação (Gibson et al., 1996). Uma solução promissora é projetar algoritmos e estruturas de dados que ignoram os sistemas de memória virtual e gerenciam explicitamente suas próprias operações de E/S, os algoritmos em memória externa. Nas subseções seguintes, será apresentado como os algoritmos e estruturas de dados em memória externa são projetados e avaliados.

1.3.1 Modelo e avaliação de algoritmos em memória externa

A maioria das linguagens de programação atuais são baseadas no modelo de memória RAM (*Random Access Memory*) (Van Leeuwen, 1990; Aho et al., 1974). Uma das principais características desse modelo é que a memória consiste de um vetor e qualquer entrada do vetor pode ser acessada com um mesmo custo.

Nesse modelo, a tarefa de mover dados para dentro e para fora da memória interna é confiada ao sistema operacional. A noção de memória virtual permite a esse vetor ser muito maior do que aquilo que possa caber na memória interna do computador. Por esse motivo, assume-se que qualquer entrada do vetor, ou qualquer área de memória, possui o mesmo tempo de acesso. Em muitos casos, essa suposição é razoável, principalmente quando o conjunto de dados a ser manipulado não é muito grande.

Porém, na prática, nem todas as áreas de memória são acessadas com o mesmo custo. Por razões econômicas, os sistemas de computadores atuais utilizam um modelo de hierarquia de níveis de memória, onde cada nível possui suas próprias características de desempenho e custo (veja Figura 1.10) (Vitter, 2001).

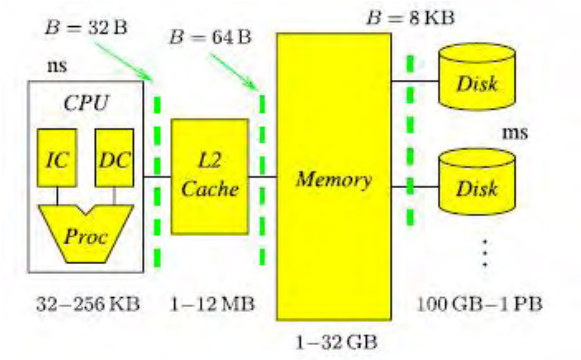


Figura 1.10: Modelo de memória hierárquico típico de sistemas uniprocessados, incluindo registradores, cache de instruções, cache de dados, cache nível 2 (L2), memória interna e discos

Nesse modelo hierárquico, o custo de acesso à memória depende do nível da memória acessada. Por exemplo, carregar um registrador pode levar uma fração de um nanosegundo (10^{-9} segundos), e acessar a memória interna pode levar alguns nanosegundos, porém o tempo para acessar dados no disco é da ordem de milissegundos (10^{-3} segundos), que é 10^6 vezes mais lento. Neste caso, o tempo necessário para acessar e transferir dados de/para a memória externa é muito maior que o tempo para o processamento interno (Toma et al., 2001; Goodrich et al., 1993).

Alguns sistemas operacionais tentam minimizar o acesso ao disco utilizando técnicas de paginação e *prefetching* para garantir que o dado esteja presente na memória interna quando for requisitado. O problema com essas técnicas é que foram projetadas para propósitos gerais e não aproveitam totalmente as características de localidade específicas de cada problema. Nesse caso, os algoritmos e estruturas de dados que não se limitam a armazenar e processar dados somente na memória interna

podem se tornar mais eficientes gerenciando explicitamente a localização e a movimentação de dados necessárias para resolver o problema a que se propõem. Esse é o caso dos algoritmos e estruturas de dados em memória externa.

Para que os algoritmos em memória externa possam controlar a transferência e a localização dos dados, é essencial que eles trabalhem sob um modelo de memória simples e acurado. Um modelo bastante utilizado, chamado de *PDM* (*Parallel Disk Model*), foi proposto por Vitter e Shriver (Vitter e Shriver, 1994). Nesse modelo, o conjunto de dados que é muito grande para caber na memória interna é tipicamente armazenado externamente em um ou mais discos magnéticos que são acessados em paralelo, obtendo assim uma largura de banda adicional. Os principais parâmetros utilizados pelo *PDM* são:

N = tamanho da entrada para o problema em itens;

M = tamanho da memória interna em itens;

B = tamanho do bloco do disco em itens;

D = número de discos independentes; onde $M < N$ e $1 \leq DB \leq \frac{M}{2}$. Uma operação de E/S é definida como o processo em que cada um dos D discos pode simultaneamente transferir um bloco de B itens contíguos.

A principal medida de desempenho para algoritmos e estruturas de dados no *PDM* é o número de operações de E/S realizadas, que geralmente é expresso em função dos limites para algumas operações fundamentais, tais como:

1. *Scanning* ou varredura de N itens contíguos do disco, cuja complexidade é $\theta(\frac{N}{B})$
2. *Sorting* ou ordenação de N itens contíguos do disco, cuja complexidade é $\theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$

É importante salientar que normalmente $scan(N) < sort(N) << N$ e, então, é significantemente melhor ter um algoritmo que realiza $sort(N)$ em vez de N operações de E/S. Assim, muitos algoritmos tentam reorganizar os dados na memória externa com o objetivo de diminuir o número de operações de E/S realizadas.

1.3.2 Ambientes para projeto de algoritmos em memória externa

Três paradigmas suportam o desenvolvimento de código eficiente para memória externa: orientação a acesso, orientação a vetor e orientação a *frameworks* (Vitter, 2001). Sistemas orientados a acesso preservam a abstração do programador de requisitar transferências de dados explicitamente. Esses sistemas oferecem uma extensão

da interface de leitura e escrita, incluindo especificações de tipos de dados e coleções para possibilitar transferências múltiplas de forma a diminuir o número de operações de entrada e saída. Sistemas orientados a vetor acessam os dados armazenados na memória externa primeiramente através de tipos de dados reconhecidos pelo compilador (tipicamente vetores). A computação externa é diretamente especificada via laços iterativos ou explicitamente através de operações paralelas nos dados. Além disso, esse sistema gerencia explicitamente as operações de E/S.

Ao invés de tratar a computação de dados em memória externa como uma sequência em que o código lê os dados, realiza a manipulação sobre eles, e os escreve de volta à memória externa, um sistema orientado a *framework* trata a computação como um processo contínuo, durante o qual o programa é alimentado com conjuntos de dados por um código externo ao programa e deixa trilhas de resultados para trás.

Dentre os três paradigmas apresentados, a orientação a *framework* é geralmente a mais utilizada por esconder do programa como a manipulação das operações de E/S é realizada, permitindo ao programa se preocupar principalmente com a resolução do problema a que se propõe. Alguns exemplos de sistemas orientados a *framework* são a TPIE e a STXXL.

A TPIE (*Transparent Parallel I/O Environment*) (Arge et al., 2002b; Arge et al., 2002c; Arge et al., 2002a; Vengroff e Vitter, 1995) é uma interface orientada a *framework* para computação em memória externa. Nesse caso, o código do algoritmo que utiliza a TPIE não precisa explicitamente chamar as rotinas que executam as operações de E/S, deve apenas especificar os detalhes funcionais do processamento desejado, e a TPIE automaticamente define e executa a sequência de movimento de dados para alcançar esse processamento.

A TPIE é um conjunto de *templates* de classes e funções em C++ que viabiliza a computação em memória externa, além de ser uma biblioteca em tempo de execução. É composta por três componentes principais: uma máquina de transferência de blocos (MTB), um gerenciador de memória (GM) e uma interface de métodos de acesso (IMA). O MTB é responsável por mover os blocos de dados de/para o disco. O GM é responsável por gerenciar a memória interna através de alocação e desalocação de acordo com as atividades do MBT e IMA. A IMA é o componente com o qual o código do algoritmo precisa interagir diretamente. Aplicações que utilizam a IMA são portáveis através das plataformas de hardware, pois não precisam lidar com os detalhes de como uma operação de E/S é desempenhada em uma máquina em particular.

A IMA disponibiliza operações como varredura, ordenação, distribuição e permutação em dados armazenados em memória externa. Além disso, ela disponibiliza

estruturas de dados como vetores, pilhas, filas, filas de prioridades e árvores de busca para organizar os dados em memória externa.

Além da TPIE, a STXXL (*Standard Template Library for XXL Data Sets*) (Dementiev et al., 2005) é uma implementação orientada a *framework* da STL (*Standard Template Library*) em C++ para manipulação de dados em memória externa. Essa biblioteca possui também 3 camadas principais, de acordo com a Figura 1.11. A camada mais baixa é composta pelas primitivas de E/S assíncronas (AIO) e abstrai os detalhes de como essas operações de E/S são desempenhadas em um sistema operacional em particular. Devido a essa camada, as aplicações implementadas utilizando a STXXL possuem portabilidade. A camada gerenciadora de blocos (BM) provê uma interface de programação que simula o *PDM*. Essa camada implementa a escrita *bufferizada* do *PDM*, *prefetching* ótimo, e *caching* de blocos.

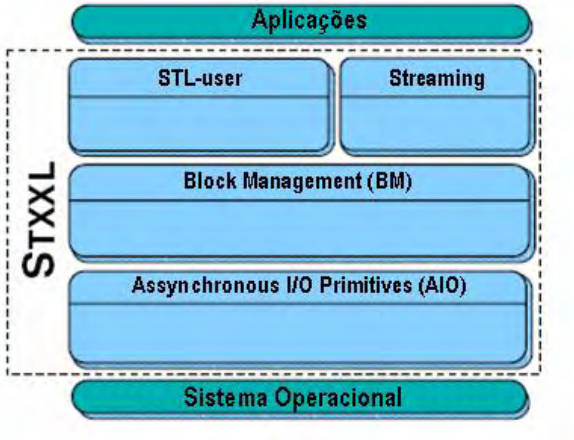


Figura 1.11: Estrutura da STXXL

O topo da STXXL é composto por dois módulos. A camada *STL-users* disponibiliza estruturas de dados em memória externa que têm quase a mesma interface, incluindo sintaxe e semântica, de suas contrapartes na STL. A camada de *Streaming* provê suporte a algoritmos em memória externa que fazem uso do paralelismo ou *pipeline*. A *STL-users* disponibiliza os mesmos algoritmos e estruturas da TPIE, mas, além disso, foi planejada para fazer uma melhor utilização dos recursos computacionais, é capaz de lidar com um maior volume de dados (dezenas de terabytes), e, segundo os autores (Dementiev et al., 2005), diminui as operações de entrada e saída necessárias para determinados algoritmos a menos da metade dos requisitos originais.

A grande maioria dos problemas envolvendo computação em memória externa, tais como problemas em GIS, computação científica, grafos e processamento de cadeias de caracteres, pode ser resolvida utilizando um número relativamente pequeno

de operações básicas como a varredura, distribuição e ordenação (Vitter, 2001), operações que as bibliotecas citadas acima disponibilizam através de suas interfaces. O problema de visibilidade apresentado nesse projeto é um desses problemas que podem ser resolvidos através de qualquer uma das duas bibliotecas apresentadas, mas devido às vantagens descritas acima, a *STXXL* foi escolhida para o desenvolvimento do algoritmo.

Capítulo 2

Trabalhos Relacionados

Neste capítulo, diversos trabalhos publicados sobre a visibilidade em terrenos, principalmente o cálculo de *viewshed*, e algoritmos em memória externa são apresentados. Durante o projeto do algoritmo em memória externa apresentado nesta dissertação, os trabalhos relacionados à visibilidade foram pesquisados com o objetivo de selecionar um algoritmo que pudesse ser adaptado para trabalhar com dados em memória externa. Já os trabalhos relacionados aos algoritmos em memória externa foram estudados para o entendimento da utilização das diversas técnicas em memória externa existentes.

2.1 Visibilidade em Terrenos

As estruturas de visibilidade, *viewsheds* e horizontes, podem ser representadas tanto de forma contínua quanto de forma discreta.

A representação contínua do *viewshed* divide cada região do terreno em suas partes visíveis e não visíveis. Essa representação para uma TIN com v vértices possui, no pior caso, uma complexidade de espaço de $O(v^2)$, uma vez que cada triângulo pode criar uma sombra nos triângulos que estão além deles (considerando a linha de visada a partir do observador) (Cole e Sharir, 1989).

Apesar de as estruturas contínuas serem geralmente calculadas a partir de TINs, alguns algoritmos foram desenvolvidos para computá-las em MDEs. Um deles é o algoritmo de Sorensen e Lanter (Sorensen e Lanter, 1993), que determina as partes visíveis e invisíveis de uma célula alvo através da união das sombras geradas nessa célula a partir de todas as células entre o observador p e a própria célula alvo.

Algoritmos para o cálculo de estruturas de visibilidade contínuas em uma TIN exploram as faces planares que a compõem. Esses algoritmos se relacionam direta-

mente com o problema de Remoção de Superfícies Ocultas (*Hidden Surface Removal* ou HSR). Em uma versão típica do problema, dado um conjunto de superfícies poliédricas no espaço tridimensional, e um observador p, o objetivo é determinar quais porções de cada superfície são visíveis a partir de p. Sendo a o número de arestas contidas no conjunto de superfícies poliédricas, se essas a arestas forem projetadas em um plano de visão, será obtido um arranjo de a segmentos geralmente interceptados. Os poliedros, quando projetados no plano de visão, geram uma decomposição poligonal do plano em regiões, sendo que apenas uma porção de uma face é visível, ou nada é visível. Essas regiões são limitadas pelas arestas dos poliedros projetados e seus vértices podem ser tanto vértices dos poliedros quanto interseções de suas arestas, veja figura 2.1.

As regiões visíveis projetadas no plano de visão para o problema de HSR englobam o *viewshed* e o horizonte de p. O *viewshed* é o conjunto de todas as regiões, enquanto o horizonte é representado pelas arestas superiores das regiões no plano de visão. As arestas superiores de um conjunto de entidades no plano mostra, para cada posição horizontal, o ponto, pertencente à alguma entidade do conjunto, que possui a máxima altitude, veja figura 2.1.

O processo mais comum para determinar as estruturas de visibilidade contínuas em uma TIN processa cada face da TIN, da mais próxima para a mais distante em relação ao observador p. Um triângulo t_1 está à frente de outro triângulo t_2 se o raio que parte de p e intercepta os dois triângulos possui a interseção com t_1 mais próxima de p do que a interseção com t_2. A abordagem do mais próximo para o mais distante é baseada no fato de que um triângulo só pode ficar escondido por triângulos que estão à sua frente.

Floriani et al. (De Floriani et al., 1989) propuseram um algoritmo incremental para calcular estruturas de visibilidade numa TIN. Nesse algoritmo os triângulos são processados na ordem do mais próximo para o mais distante do observador p através da construção incremental de um polígono em forma de estrela ao redor de p, começando pelo triângulo contendo p e adicionando os outros triângulos um a um, até que todos os triângulos tenham sido adicionados ao polígono. Triângulos fora do polígono em forma de estrela mas adjacentes a ele são os candidatos para o próximo processamento. Se um triângulo t puder ser incluído no polígono mantendo sua forma de estrela, então ele é processado. As porções visíveis do triângulo t são determinadas através da projeção do horizonte correspondente ao polígono já processado no plano de suporte de t. Esse algoritmo possui uma complexidade de $O(v^2\alpha(v))$, onde v é o número de vértices da TIN e α é o inverso da função de Ackermann (Cole e Sharir, 1989) e na prática pode

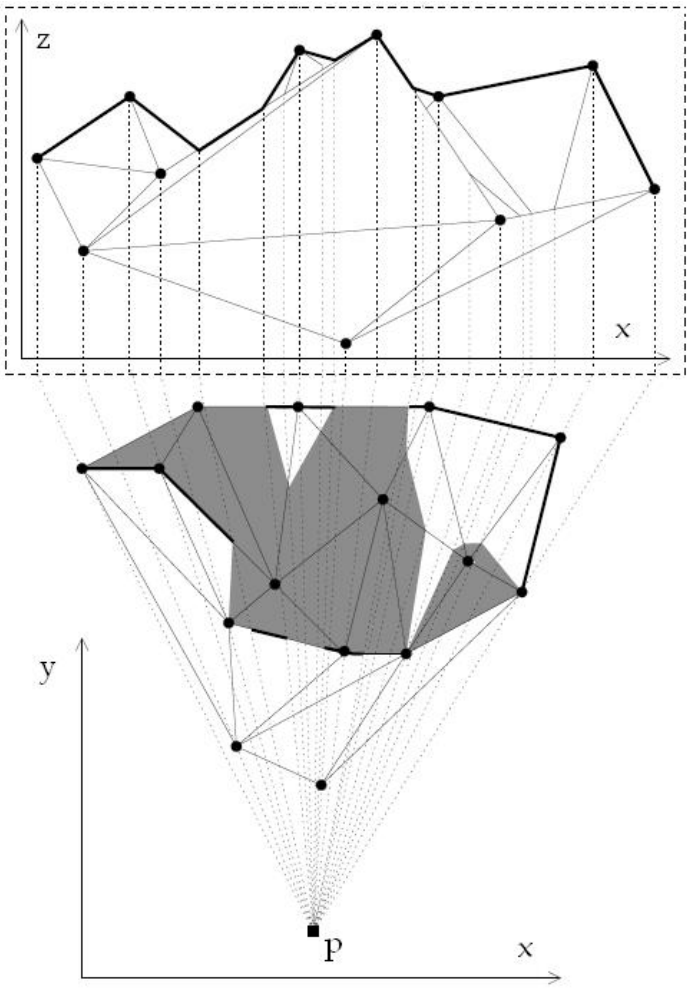


Figura 2.1: *Viewshed* de p em uma TIN no plano xy e projetado em xz como o plano de visão de p. Áreas escuras não são visíveis por p. Arestas mais grossas marcam o horizonte no plano xy e as arestas superiores no plano xz.

ser considerado como uma constante. Uma implementação paralela desse algoritmo, alcançada através da divisão da TIN em seções radiais, foi descrita em (De Floriani et al., 1994).

Outros algoritmos também baseados na abordagem incremental foram desenvolvidos para serem sensíveis à saída (*output-sensitive*), isto é, a complexidade, no pior caso, depende do tamanho da saída gerada pelo algoritmo. Isso pode ser alcançado através da utilização de estruturas mais complexas para armazenamento do horizonte a cada triângulo processado (Preparata e S., 1992; Reif e Sen, 1988). A complexidade desses algoritmos é $O((v+d)\log^2 v)$, onde v é o número de vértices da TIN e d é o tamanho da saída.

Reif e Sen (Reif e Sen, 1988) propuseram um algoritmo paralelo sensível à

saída para o problema de HSR em uma TIN ordenável. Uma TIN é ordenável quando é possível ordenar seus triângulos em relação ao observador do mais próximo para o mais distante. Nem toda TIN é ordenável, mas aquelas construídas através da triangulação de Delaunay possuem essa característica (De Floriani et al., 1991). Além disso, de acordo com (Cole e Sharir, 1989), uma triangulação pode se tornar ordenável dividindo-se alguns de seus triângulos.

O algoritmo de Reif e Sen assume que os processadores são livres e, portanto, podem ser requisitados em tempo de execução. As projeções das arestas do terreno no plano xy são recursivamente divididas em subconjuntos de mesmo tamanho de forma que cada subconjunto esteja à frente de outro. As arestas superiores de cada subconjunto são computadas em paralelo. Além disso, também são computadas em paralelo as interseções de cada uma das arestas com as arestas superiores do subconjunto em frente a elas.

Um algoritmo de divisão e conquista foi projetado por Katz, Overmars e Sharir (Katz et al., 1991), para resolver o HSR. Na etapa de divisão do algoritmo, os objetos representando o terreno são associados às folhas de uma árvore binária balanceada T de tal forma que o caminhamento na árvore da esquerda para a direita respeite a ordem dos objetos do mais próximo para o mais distante. Para a etapa de conquista, o caminhamento em T é realizado duas vezes. No primeiro caminhamento (*bottom-up*), para cada nó N é computada a união $U(N)$ da imagem de todos os objetos associados a folhas da subárvore cuja raiz é N. No segundo caminhamento (*top-down*), é calculada, para cada nó N, a região visível $V(N)$ de $U(N)$. No fim da execução do algoritmo, as regiões visíveis de cada objeto são representadas pelos conjuntos V das folhas de T. Para uma TIN ordenável, esse algoritmo possui uma complexidade de $O((d + v\alpha(v)) \log v)$, onde v é o número de vértices da TIN, d é o tamanho da saída e α é o inverso da função de Ackermann. Uma versão paralela para esse algoritmo foi proposta em (Teng et al., 1995) considerando um nível da árvore de cada vez e desempenhando em paralelo as operações relacionadas aos nós desse nível.

As estruturas de visibilidade discretas são baseadas na interseção entre a linha de visada, LOS, partindo do observador e as células (ou pontos) dos modelos de terreno. Estruturas discretas para TINs podem ser obtidas considerando a marcação de cada triângulo ou vértice do terreno como visível ou não-visível. Nesse caso, para uma TIN com v vértices uma marcação é feita para cada vértice ou triângulo e, portanto, a complexidade de espaço para essa estrutura é de $O(v)$ (Lee, 1991).

O algoritmo descrito por Lee (Lee, 1991) calcula a estrutura de visibilidade discreta para uma TIN através de uma abordagem do mais próximo para o mais

distante, similar àquela usada em (De Floriani et al., 1989). Nesse algoritmo, cada triângulo é considerado visível somente se todos os seus vértices são visíveis pelo observador p.

Apesar de existirem algoritmos para estruturas de visibilidade discretas em TINs, essas são geralmente usadas para MDEs. O algoritmo proposto por Shapira (Shapira, 1990) traça uma LOS partindo do observador p até um ponto q chamado de alvo. O algoritmo caminha ao longo dessa LOS e, se uma interseção entre a LOS e um ponto do terreno for encontrada antes de q ser atingido, q é dito não visível. Caso contrário, q é dito visível. Esse método possui a desvantagem de desempenhar muitos cálculos redundantes, uma vez que as linhas de visada para diferentes pontos da matriz podem se sobrepor parcialmente.

Para tentar diminuir essa redundância, o método projetado por Blelloch (Blelloch, 1990) considera somente as linhas de visada que unem o observador p a pontos situados no lado mais distante (*far side*) da região ou célula alvo. Esse algoritmo é baseado na seguinte premissa: dado um observador p, um conjunto mínimo de linhas de visada, cobrindo todos os pixels na região alvo, é obtido através de raios que partem de p e chegam aos pixels do lado mais distante da região. O lado mais distante é uma aresta a na fronteira da região alvo tal que tanto a região quanto p pertencem ao mesmo lado de a, veja figura 2.2. O algoritmo determina a visibilidade de cada célula ao longo da LOS através da tangente de p para o terreno. Uma aproximação para a estrutura discreta, bem como uma redução de complexidade, pode ser alcançada através da utilização de apenas um subconjunto de linhas de visada ao redor de p.

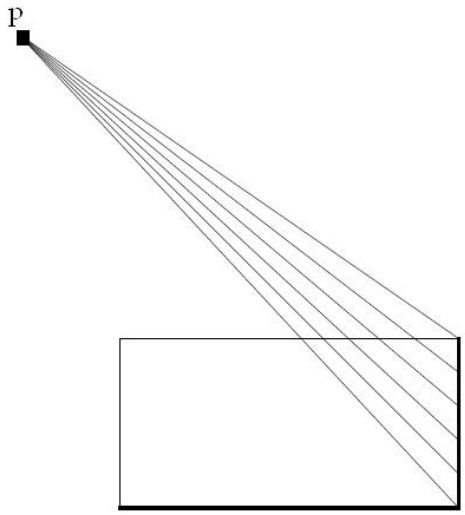


Figura 2.2: Lados mais distantes (linhas espessas) de uma região alvo, e as linhas de visada do observador p para uma das regiões

Fisher (Fisher, 1996) mostra que a abordagem por linha de visada pode ser usada para computar *viewsheds* estendidos sem nenhum *overhead* computacional. Van Kreveld (Van Kreveld, 1996) computa *viewsheds* estendidos através de uma varredura radial. Esse algoritmo considera uma linha de visada girando ao redor do observador p e, durante o giro, a informação de visibilidade é computada para todas as células interceptadas pela linha. A varredura é parada toda vez que ocorre um evento que pode modificar as informações; por exemplo, quando ocorre a interseção de novas células da matriz. A complexidade no pior caso é dada por $O(n \log n)$ para um MDE de $\sqrt{n} \times \sqrt{n}$ células.

Franklin e Ray (Franklin e Ray, 1994) propuseram um algoritmo aproximado muito eficiente para computar o *viewshed* discreto e os *offsets* dentro de uma distância r do observador em um MDE. O algoritmo processa as células através de anéis concêntricos, tendo início no anel adjacente ao observador p. A visibilidade de um ponto q pertencente ao i-ésimo anel é determinada pelos *offsets* de dois pontos pertencentes ao anel anterior, q_1 e q_2, tal que a linha pq passa entre eles. A elevação do segmento q_1q_2 é interpolada e usada para estimar a elevação da linha de visada sobre q. O método é aproximado pois q pode ser sombreado por um ponto em um anel mais interno e, então, não ser verificado pelo algoritmo.

Uma variante desse algoritmo, proposta em (Franklin, 2002), computa de forma exata o *viewshed* discreto em um MDE. Dado um terreno representado por uma matriz de elevação T, $n \times n$, e dado um observador p em T, o algoritmo computa o *viewshed* de p em um círculo de raio r (raio de interesse) centrado em p. O algoritmo desempenha uma varredura radial do círculo usando um raio, uma LOS, começando em p e atingindo as células da borda do círculo. O algoritmo caminha ao longo de cada LOS para determinar se cada posição do terreno na LOS é visível por p ou não. Uma posição do terreno q é visível por p se a LOS não intercepta nenhuma posição cuja altura é maior que a de q. Esse método foi utilizado como base para o algoritmo proposto nesse trabalho. Maiores informações sobre esse algoritmo podem ser obtidas no Capítulo 3.

Alguns algoritmos utilizam paralelismo para computar *viewsheds* discretos em MDEs, devido à topologia regular desse modelo de terreno. O método proposto em (Mills et al., 1992) é uma versão paralela do método de Shapira (Shapira, 1990). Nesse algoritmo, todas as linhas de visada partindo do observador são processadas em paralelo. Teng et al. (Teng et al., 1995) propuseram um algoritmo que, assim como Blelloch em (Blelloch, 1990), consideram apenas linhas de visada que partem do observador e chegam a um ponto no lado mais distante da região alvo. Todas as linhas

de visada correspondentes ao lado mais distante são processadas em paralelo. Rallings et. al (Rallings et al., 1998) projetaram um algoritmo para computar a visibilidade de todos os vértices do MDE utilizando um *cluster* de estações de trabalho. Cada *cluster* conhece todo o MDE mas computa a visibilidade de apenas um subconjunto dos observadores a serem considerados. Nesse trabalho, os autores realizaram diversos experimentos para o particionamento dos observadores entre os processadores, utilizando tanto critérios estáticos quanto dinâmicos, e os compararam com relação à eficiência e balanceamento de carga.

O horizonte para uma TIN pode ser obtido como um resultado de algoritmos para cálculo de estruturas de visibilidade contínuas. Por exemplo, como visto anteriormente, o horizonte é mantido como uma estrutura auxiliar na abordagem incremental do mais próximo para o mais distante. O horizonte também pode ser computado como a aresta superior do conjunto de segmentos obtidos pela projeção das arestas do terreno em um plano de visão. A complexidade herdada para problema é de $\theta(v \log v)$, onde v é o número de vértices da TIN, uma vez que esse problema é equivalente ao problema de ordenação.

Algoritmos que calculam o horizonte através das arestas superiores utilizam abordagens dividir para conquistar ou incremental. Algoritmos por divisão e conquista possuem complexidade de $O(v\alpha(v) \log v)$ (Atallah, 1983), ou de $O(v \log v)$ no pior caso(Hershberger, 1989). Um algoritmo incremental possui a complexidade $O(v^2\alpha(v))$ e uma versão mais refinada (De Floriani e Magillo, 1995) possui uma complexidade de $O(v\alpha(v) \log v)$. Em todos esses valores de complexidade v é o número de vértices da TIN.

Stewart (Stewart, 1998) computa um horizonte aproximado para todos os pontos do modelo do terreno. O método processa somente os vértices do modelo desconsiderando a estrutura de conectividade abaixo desses vértices na matriz (MDE) ou na triangulação (TIN). Para cada observador, s setores radiais são traçados e considera-se que o horizonte tem uma elevação constante dentro de cada setor. O algoritmo processa o i-ésimo setor para todos os observadores ao mesmo tempo. A complexidade desse método no pior caso é de $O(sv \log^2 v)$ para um terreno de v vértices.

2.2 Algoritmos em memória externa

Os algoritmos em memória externa podem ser divididos em duas categorias: problemas em batelada e problemas *on-line* (Vitter, 2001). Os problemas em batelada são aqueles em que nenhum pré-processamento é realizado nos dados; o conjunto total de

dados precisa ser processado. Isso é realizado frequentemente enviando-se os dados para a memória interna em um ou mais passos. Exemplos de problemas desse tipo são os problemas de ordenação, problemas da geometria computacional e da teoria dos grafos. Nos problemas *on-line*, a computação é realizada em resposta a uma série de operações de consulta. Uma técnica comum para esse tipo de problemas é organizar os dados através de um índice hierárquico. Dessa forma, apenas uma pequena porção dos dados precisa ser examinada para responder a cada uma das consultas. Problemas *on-line* envolvem o projeto de estruturas de dados com o objetivo de oferecer bom desempenho nas consultas sobre os dados.

Um dos problemas em batelada mais importantes e um dos primeiros a ter sua versão em memória externa foi a ordenação. Nos algoritmos em memória externa, operações de ordenação contribuem com um percentual significante do uso do computador (Knuth, 1998); por esse motivo, grande esforço dos pesquisadores foi concentrado em resolver esse problema de forma eficiente em memória externa. Dentre os diversos trabalhos envolvendo o problema de ordenação, Vitter e Shriver (Vitter e Shriver, 1994) e DeWitt et al. (DeWitt et al., December 1991) desenvolveram algoritmos aleatórios para realizar ordenação utilizando o paradigma de distribuição enquanto Aggarwal e Plaxton (Aggarwal e Plaxton, 1994) e Barve et al. (Barve et al., 1997; Barve e Vitter, March/April 2002) desenvolveram versões em memória externa do *mergesort* (Knuth, 1998).

Goodrich et. al (Goodrich et al., 1993) foram os primeiros a considerarem o projeto de algoritmos de geometria computacional em memória externa. Eles usaram suas técnicas para desenvolver uma versão do paradigma *plane sweep* em memória externa que recebeu o nome de *distribution sweep* por utilizar técnicas do paradigma de distribuição. Os autores ainda utilizaram o *distribution sweep* para projetar algoritmos ótimos para um grande número de problemas, incluindo interseção de segmentos de linha ortogonais, determinação dos vizinhos mais próximos, interseção de pares de retângulos e consultas em batelada de intervalos. Eles ainda apresentaram técnicas para construção de árvores B persistentes - árvores B capazes de retornar versões prévias da estrutura de dados - e filtragem em batelada - técnica para consultas em estruturas de dados que representam as estruturas em blocos de disco de tal forma que as consultas possam ser desempenhadas com um número mínimo de operações de E/S. Além dessas técnicas, eles apresentaram algoritmos em memória externa para a construção de cascos convexos em duas e três dimensões através do paradigma de *marriage-before-conquest* (Kirkpatrick e Seidel, 1986). Todos esses algoritmos possuem complexidade de $O((N/B + K/B)\log_{M/B} N/B + T/B)$, onde M, N e B são

os parâmetros definidos na Seção 1.3.1, K é o número de operações de consulta no problema e T é o número de itens na solução do problema, sendo que $K = 0$ para problemas que não envolvem consultas e $T = 1$ para problemas em que a solução possui tamanho fixo.

Além dos problemas base da geometria computacional citados acima, alguns problemas relacionados à hidrologia foram resolvidos de maneira eficiente em memória externa. Em (Toma et al., 2001), Arge e Vitter apresentam um algoritmo com complexidade de $O(N/B \log_{M/B} N/B)$ operações de E/S para o cálculo da acumulação de fluxo em bacias hidrográficas, além de algoritmos em memória externa para a busca em largura e para o problema de caminho mais curto com única origem (um algoritmo para o cálculo do caminho mais curto com múltiplas origens é apresentado em (Hazel et al., 2006)). Arge et al. (Arge et al., 2003) apresentam um algoritmo para a computação da direção do fluxo em bacias hidrográficas com uma complexidade de $O(N/B \log_{M/B} N/B)$ operações de E/S. Ainda nesse trabalho Arge et al. apresentam o *TERRAFLOW*, o primeiro software para análises de terrenos projetado e otimizado para grids com grande volume de dados, cujos fundamentos teóricos são formados pelos dois algoritmos de computação de fluxo citados acima. Em (Arge et al., 2006), Haverkort et al. propõem um algoritmo para decomposição hierárquica de MDEs em bacias hidrográficas em memória externa através da computação do índice de Pfafstetter (Verdin e Verdin, 1999) de cada célula do grid. O algoritmo computa os índices de N células do terreno com $O(N/B \log_{M/B} N/B)$ operações de E/S.

Outra aplicação para os algoritmos de geometria computacional em memória externa foi apresentada em (Haverkort et al., 2007) para o cálculo de *viewshed* com complexidade de $O(N/B \log_{M/B} N/B)$. Nesse artigo, Haverkort et al. fazem uma adaptação do método de Van Kreveld (Van Kreveld, 1996) para manipular a matriz de elevação do terreno em disco. Mais informações sobre esse algoritmo podem ser encontradas no Capítulo 4.

Capítulo 3

Algoritmo para Cálculo do *Viewshed* em terrenos armazenados em memória externa

Este capítulo descreve dois algoritmos para o cálculo do *viewshed* em torno de um observador: um deles considerando um terreno armazenado em memória interna, o algoritmo de Franklin e Ray, e outro considerando dados em memória externa, o algoritmo EMVS *External Memory Viewshed* - uma adaptação do método de Franklin e Ray e a principal contribuição desta dissertação.

3.1 O método de Franklin e Ray

O algoritmo proposto por Franklin e Ray (Franklin, 2002) é uma versão bem simples para o cálculo do *viewshed* em torno de um observador à partir de um MDE ou matriz de elevação. Esse algoritmo é reconhecidamente simples e possui bom desempenho em terrenos pequenos, da ordem de KBytes, conforme os resultados apresentados em (Franklin, 2002).

Dado um terreno representado por uma matriz de elevação T de dimensões $n \times n$, um ponto p em T e um raio de interesse r, o algoritmo desenvolvido por Franklin e Ray (Franklin e Ray, 1994) calcula o *viewshed* de p restrito a um círculo de raio r centrado em p. De forma geral, o algoritmo percorre o círculo usando linhas de visada (LOS), que têm início em p e fim em cada uma das células de borda do círculo. Através do caminhamento por cada LOS, ocorre a classificação das células da matriz pertencentes à LOS como visíveis a partir de p ou não.

Para simplificar a identificação das diversas LOS e consequentemente a varre-

dura do círculo, o algoritmo utiliza um quadrado, chamado de quadrado de varredura, que circunscreve o círculo, tem lado $2r$ e é centrado em p. Inicialmente, todas as células que estão dentro do quadrado são consideradas não visíveis e as linhas de visada são definidas e processadas uma a uma, conectando p a cada ponto da borda do quadrado de varredura, veja Figura 3.1.

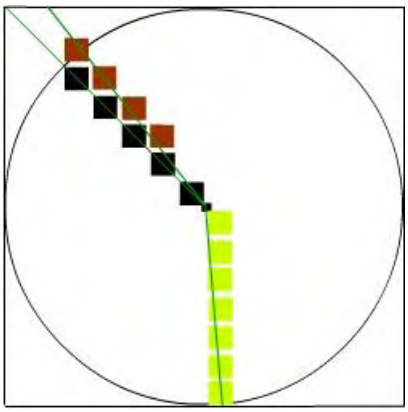


Figura 3.1: Cálculo do *viewshed*: linhas de visada com suas respectivas rasterizações

A identificação dos pontos que fazem parte de uma LOS é realizada através da rasterização de linhas. O processo de rasterização utilizado no algoritmo determina as células que compõem a LOS inicialmente através da identificação da coordenada, x ou y, de maior variação entre o observador p e a célula f da borda do quadrado. Isso é, considerando (p_x, p_y) as coordenadas de p e (f_x, f_y) as coordenadas de f, e considerando também que a inclinação α da reta que passa por p e f é menor que 45^o, a coordenada x terá maior variação se o valor de $|f_x - p_x|/|f_y - p_y|$ for maior que 1, caso contrário, a coordenada y terá maior variação. Para outros valores de α, são realizados cálculos semelhantes para a identificação da coordenada de maior variação. O algoritmo faz a coordenada de maior variação variar entre p e f por incrementos de uma unidade e obtém a outra coordenada através da equação da reta que passa pelos pontos p e f. O procedimento de determinação da coordenada de maior variação é realizado para evitar que haja descontinuidade na representação da LOS, veja Figura 3.2.

Após a identificação de cada célula de uma LOS, o algoritmo precisa determinar se essa célula é visível ou não por p. Por questões de desempenho, o método utilizado para determinação de visibilidade entre p e os pontos da LOS difere do método descrito na Seção 2.2.2. Para cada linha de visada l, o algoritmo inicia pelo ponto p definindo a inclinação de l como $-\infty$ (isto é, um número negativo grande) e prossegue pelas células que compõem l até a borda do círculo.

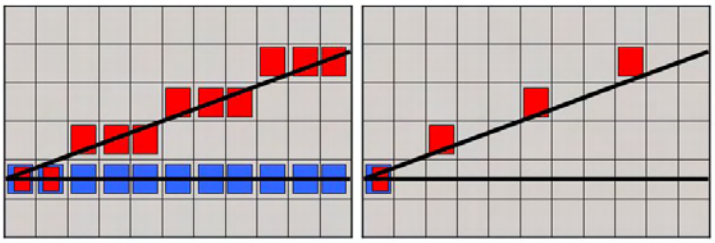


Figura 3.2: Pontos escuros mostram a rasterização das linhas apresentadas na Figura : (a) incrementando x e calculando y e (b) incrementando y e calculando x

A cada célula c da LOS l, o algoritmo calcula a inclinação da linha de visada l' que liga p a c. Se a inclinação de l' é maior que a inclinação de l então a inclinação de l é atualizada com o valor da inclinação de l'. Isso é, supondo que a inclinação de l seja h, que a elevação de p seja e_p, que a elevação de c seja e_c e que a coordenada de maior variação entre p e c seja x, então a inclinação h' de l' é dada por

$$h' = (e_c - e_p)/\left|c_x - p_x\right| \tag{3.1}$$

Daí, se $h < h'$ então a inclinação de l é atualizada para h' e a célula c é considerada visível. Por outro lado, se $h > h'$, a inclinação de l é preservada assim como o estado de c.

Apesar do algoritmo utilizar o quadrado de varredura para facilitar o traçado da LOS, o mesmo utiliza o círculo determinado pelo raio de visão como limite para as células a serem processadas. Por isso, a cada célula c que compõe l, a distância entre o observador p e a célula c é calculada e c é processada se, e somente se, essa distância for menor que o raio r, ou seja, supondo que a célula c e o observador p possuam as coordenadas (c_x, c_y) e (p_x, p_y), respectivamente, então a célula c é processada se

$$\sqrt{(c_x - p_x)^2 + (c_y - p_y)^2} \leq \sqrt{r} \tag{3.2}$$

Com isso, as células entre o círculo e o quadrado são ignoradas pelo algoritmo.

Algumas vezes, a própria borda do terreno é o limite para o processamento das células, pois o observador pode estar próximo à borda fazendo com que o raio de interesse ultrapasse as bordas do terreno. Por esse motivo, durante o processamento de cada célula, além de verificar se a mesma pertence ao círculo conforme a equação 3.2, também é necessário verificar se a célula encontrada está entre os limites do terreno.

Devido ao uso do quadrado de varredura para traçado das linhas de visada,

algumas células, principalmente aquelas mais próximas do observador, são interceptadas por mais de uma LOS, veja Figura 3.3. O processamento dessas células ocorre da mesma forma citada acima, porém essas células podem ser visíveis por p em uma determinada LOS e não visíveis em outras. Nesse caso, a célula que é visível por pelo menos uma LOS, é considerada visível pelo algoritmo.

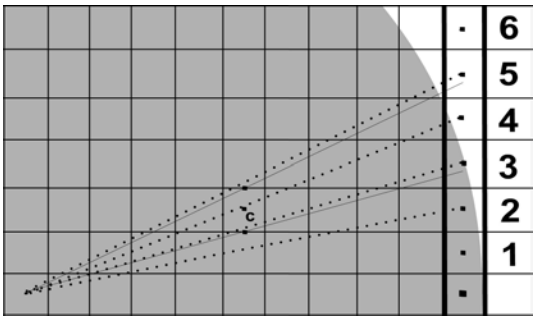


Figura 3.3: Célula interceptada por mais de uma LOS. Célula c é intereceptada pelas linhas 3 e 4.

Após o processamento de todas as células no quadrado de varredura, o resultado final do algoritmo é uma matriz onde cada célula representa um ponto do terreno e cujo conteúdo é composto por 1 ou 0, indicando se a célula é visível ou não pelo observador p, respectivamente.

Esse algoritmo é muito eficiente quando o terreno pode ser armazenado na memória interna. Porém, quando o terreno é muito grande, há a necessidade de manipular o terreno em memória externa e nesse caso, o elevado número de acessos aleatórios torna inviável o uso desse algoritmo.

3.2 Algoritmo em Memória Externa - EMVS

Esta seção apresenta uma adaptação do algoritmo apresentado na seção anterior para o cálculo do *viewshed* em terrenos armazenados em memória externa, o EMVS (*External Memory Viewshed*).

A idéia que permeou a elaboração do algoritmo EMVS foi a reorganização dos elementos da matriz de elevação de forma a diminuir os acessos aleatórios aos dados, a principal causa da ineficiência do algoritmo de Franklin e Ray. A primeira operação realizada pelo algoritmo sobre a matriz de elevação é a determinação da ordem de processamento de cada célula da matriz e a posterior ordenação da matriz de acordo com a ordem de processamento das células. Com essa reorganização, a primeira célula a ser processada pelo algoritmo estará na primeira linha e primeira coluna da matriz,

a segunda célula a ser processada estará na primeira linha e segunda coluna, e assim por diante, obedecendo uma ordem linha-coluna e permitindo o acesso direto a cada célula durante a determinação da visibilidade.

Mais especificamente, cada célula c da matriz é associada a um índice i que indica em qual momento do algoritmo a célula deverá ser processada. Isso é, se uma célula c está associada ao índice i, então c será a i-ésima célula a ser processada pelo algoritmo. Conforme explicado na seção anterior, algumas células da matriz são processadas mais de uma vez, por serem interceptadas por mais de uma linha de visada. Para incluir essas situações na matriz e garantir que a célula continue sendo processada o mesmo número de vezes que no algoritmo de Franklin e Ray, a solução adotada foi representar a matriz como uma lista de pares (célula, índice) sendo que uma mesma célula pode aparecer em vários pares mas com índices diferentes.

Para criar a lista com os pares (célula, índice), as linhas de visada, que se iniciam no observador p, são numeradas em sentido anti-horário, sendo a LOS horizontal com início em p e fim na última célula do raio de visão à direita de p, a LOS de número 0, veja Figura 3.4. Com as linhas de visada numeradas, o algoritmo lê as células da matriz sequencialmente a partir do arquivo em disco e, para cada célula c, determina o número de todas as linhas de visada que interceptam aquela célula, uma vez que o índice depende da linha de visada que intercepta a célula.

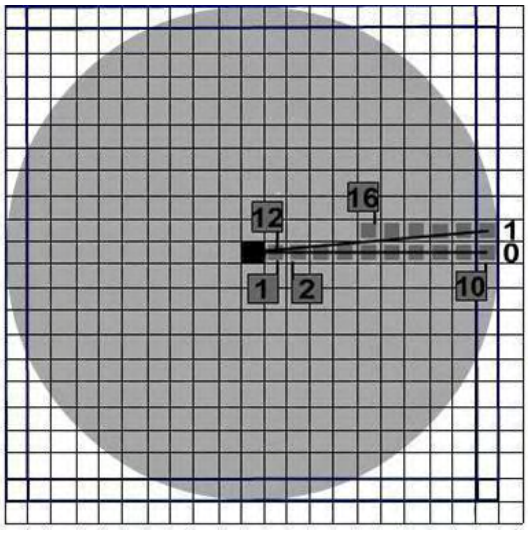


Figura 3.4: Numeração das linhas de visada com início no observador p em sentindo anti-horário

Para obter as linhas de visada que interceptam cada célula, o algoritmo utiliza o fato de que uma célula não é unidimensional e processa o terreno em octantes devido às diferentes características de cada uma dessas partes do terreno. A forma

de processamento descrita a seguir será relativa às células do primeiro octante da matriz, para os demais octantes é adotado um processo análogo, fazendo-se os devidos ajustes.

Seja c uma célula no primeiro octante da matriz de elevação e s o valor da largura do lado de c em pixels. Seja a uma linha de visada no mesmo octante, consequentemente a inclinação de a está entre 0^o e 45^o. Sejam (c_x, c_y) as coordenadas de c, então, adotando uma estratégia semelhante ao algoritmo de Bresenham (Bresenham, 1965), consideramos que a linha de visada a intercepta a célula c se a passa entre os pontos $(c_x, c_y - 1/2s)$ e $(c_x, c_y + 1/2s)$. Mais precisamente, seja q=(q_x, q_y) o ponto de interseção entre c e a, então, a intercepta c se $c_y - 1/2s \leq q_y \leq c_y + 1/2s$. Nesse caso, a linha tracejada na Figura 3.5 não interceptará a célula c e sim a célula imediatamente acima de c.

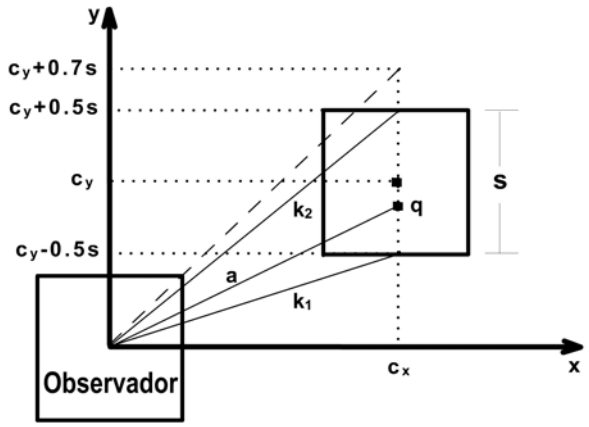


Figura 3.5: Linhas de visão interceptando uma célula

Como o objetivo do algoritmo EMVS é determinar quais linhas de visada passam por c, então, todas as linhas de visada que passam entre os pontos $(c_x, c_y - 1/2s)$ e $(c_x, c_y + 1/2s)$, interceptam c. Para isso, o algoritmo inicialmente determina as duas células da borda do quadrado de varredura que são tocadas pelas linhas de visada que passam pelos pontos $(c_x, c_y - 1/2s)$ e $(c_x, c_y + 1/2s)$. Essas células podem ser vistas como a primeira e a última células tocadas pelas linhas de visada que passam por c durante a varredura radial em sentido anti-horário, veja Figura 3.6.

De forma simplificada, as linhas de visada que interceptam c são aquelas que estão entre as duas linhas que conectam o observador aos pontos $(c_x, c_y - 1/2s)$ e $(c_x, c_y + 1/2s)$. Sejam k_1 e k_2 os números dessas duas linhas, respectivamente. Os números de todas as linhas que interceptam c são dados pelas células cujo centro

estejam entre k_1 e k_2. Na Figura 3.6, por exemplo, a célula c é interceptada pelas linhas de visada 5, 6, 7, e 8.

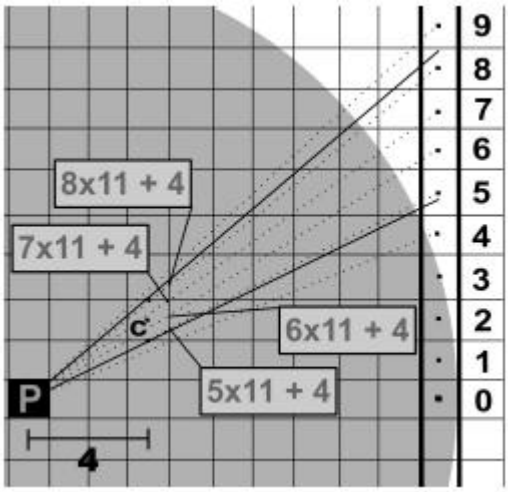


Figura 3.6: Projeção de LOS e cálculo dos índices de uma célula

Mais precisamente, sejam t_1 e t_2 a primeira e a última células na borda da região de interesse que definem o intervalo contendo as linhas de visada que interceptam a célula c. As células t_1 e t_2 podem ser determinadas da seguinte forma: sejam α_1 e α_2 as inclinações das linhas de visada que passam por $(c_x, c_y - 1/2)$ e $(c_x, c_y + 1/2)$, respectivamente. Então, considerando que o observador p está centrado no ponto $(0, 0)$, temos que

$$\alpha_1 = (c_y - 1/2s)/c_x \tag{3.3}$$

$$\alpha_2 = (c_y + 1/2s)/c_x \tag{3.4}$$

Assim, as coordenadas das células t_1 e t_2 são $(r, r\alpha_1)$, $(r, r\alpha_2)$, respectivamente, onde r é o raio de interesse. Daí, os números da primeira e da última linhas de visada que passam por c podem ser identificados pelo valor inteiro obtido através do arredondamento da coordenada y das células t_1 e t_2 determinadas, isto é, se $t_1 = (t_{1x}, t_{1y})$ e $t_2 = (t_{2x}, t_{2y})$, então a primeira linha de visada que intercepta c é a t_{1y}-ésima LOS do quadrado de varredura e a última linha de visada que intercepta c é a t_{2y}-ésima LOS do quadrado de varredura. Consequentemente, as linhas de visada que interceptam c são aquelas que estão entre a t_{1y}-ésima e a t_{2y}-ésima LOS ou, mais especificamente, entre a LOS de número $r\alpha_1$ e $r\alpha_2$.

Uma vez identificado o número das linhas de visada que interceptam a célula

c, podemos agora determinar o índice associado a c considerando cada uma dessas LOS. Para isso, basta observar que, no algoritmo de EMVS, assim como no algoritmo de Franklin e Ray, as células começam a ser processadas a partir do observador p seguindo através da linha de visada corrente até a última célula pertencente ao raio de interesse. Depois volta-se ao observador p e segue pela próxima linha de visada até que o algoritmo tenha passado por todas as células do quadrado de varredura. A partir dessa ordem de processamento, podemos associar ao observador p o índice, ou ordem de processamento, 1, pois p será sempre a primeira célula a ser processada pelo algoritmo. À próxima célula à direita de p pertencente à LOS de número 0 podemos associar o índice 2 e à última célula da LOS de número 0 podemos associar o índice r. Ao chegar ao fim da LOS de número 0, o algoritmo retorna à celula correspondente ao observador p que será novamente processada, à qual deve ser associado o índice $r+1$, veja Figura 3.4. Portanto, dada uma célula c e uma linha de visada n que a intercepta, o índice de c associado a n pode ser determinado pela fórmula

$$i = n \times k + d \tag{3.5}$$

onde k é uma constante que indica o número de células em cada raio e d é a distância entre os pontos c e p em número de células.

A distância d usada pela equação (3.5) corresponde à maior variação entre as coordenadas de c e do observador p, ou seja, se $c = (c_x, c_y)$ e $p = (p_x, p_y)$, então

$$d = maior(|c_x - p_x|, |c_y - p_y|)$$

Uma vez encontradas todas as linhas de visada que interceptam a célula c, para cada uma dessas linhas o índice de c correspondente é determinado pela equação (3.5) e inserido em uma estrutura L, uma lista em memória externa, que armazena uma estrutura composta pelo índice, coordenadas e elevação da célula c. A Figura 3.6 mostra como os índices da célula c são calculados para cada uma das 4 linhas de visada que interceptam c. Após todas as células do quadrado de varredura terem sido processadas, isto é, todas as células terem sido lidas e seus índices de processamento computados, a lista L é ordenada usando os índices das células como chave de ordenação.

A determinação da visibilidade das células é realizada por um algoritmo similar ao de Franklin e Ray, descrito na seção anterior, que, porém, lê os dados de elevação diretamente de L. Durante o processamento, as células classificadas como visíveis são armazenadas em uma estrutura de outra lista em memória externa L' contendo apenas as coordenadas da célula, uma vez que as informações de índice e elevação

são desnecessárias a partir desse momento. Por fim, quando todas as células de L já foram processadas e as visíveis já foram inseridas em L', a lista L' é ordenada lexicograficamente pelas coordenadas x e y e, a partir dessa ordenação, uma matriz é preenchida com 1 e 0 indicando as posições visíveis e não visíveis, respectivamente.

3.2.1 Implementação

A implementação do EMVS foi realizada em C++ seguindo os mesmos padrões do projeto do algoritmo de Franklin e Ray. O programa possui uma interface com a biblioteca STXXL, também implementada em C++, usada para criação e manipulação das estruturas de dados em memória externa.

A idéia básica do algoritmo EMVS é, então, bastante simples: reestruturar a matriz de elevação de forma a ordenar os elementos de acordo com a ordem de processamento, para que, a cada passo, o algoritmo tenha acesso eficiente à próxima célula a ser processada. Para isso, o algoritmo inicialmente lê cada uma das células da matriz de elevação e determina todos os índices, indicando quando a célula é processada, e insere todos os pares (célula, índice) em uma lista L, que chamaremos de lista de varredura. Após a leitura de todos os elementos da matriz de elevação, L é ordenada utilizando os índices como chave de ordenação. Após esse passo, o algoritmo calcula o *viewshed* processando cada célula de L, determinando sua visibilidade, e insere as células visíveis em uma outra lista L', a lista de pontos visíveis. Ao final do processamento, o algoritmo ordena L' de acordo com as coordenadas de cada célula e usa essa lista para construir a matriz que representará o *viewshed* da seguinte forma: se uma célula está em L' sua posição na matriz é preenchida com o número 1, caso contrário, sua posição é preenchida com 0.

Na implementação, a função principal do algoritmo recebe 7 parâmetros: o nome do arquivo que armazena a matriz de elevação, o tamanho em bytes que ocupa cada elevação (elsize), o número de linhas da matriz (nrows), a posição do observador (ox e oy), a altura do observador acima do terreno (observerElev) e o tamanho do raio de interesse (roi). Em posse esses dados, o algoritmo executa os seguintes passos:

1. Geração da lista de varredura
2. Determinação dos índices das células
3. Cálculo de *viewshed*
4. Geração da matriz de *viewshed*

O conteúdo de cada um desses passos, que são mapeados em funções na implementação do algoritmo, será mostrado detalhadamente nas próximas seções. Primeiramente, entretanto, iremos apresentar uma breve descrição sobre a estrutura de dados vector da biblioteca STXXL, utilizada pelo EMVS para implementar as listas em memória externa, e sobre o algoritmo de ordenação utilizado nessas estruturas, o stxxl :: sort .

Estrutura de dados stxxl :: vector

O tipo vector na STL (std :: vector) é um vetor que suporta acesso aleatório aos elementos e que possui um tempo constante para as operações de inserção e remoção dos elementos no final do vetor. Além disso, o tamanho do std :: vector pode variar dinamicamente. Na STXXL o stxxl :: vector é a estrutura mais comum da biblioteca e mantém compatibilidade com os métodos básicos do std :: vector. O vetor na STXXL permite o uso de iteradores.

O stxxl : vector é organizado como uma coleção de blocos na memória externa. O acesso a esses blocos externos é realizado através de uma *cache* associativa que consiste de uma quantidade fixa de páginas[1] em memória, veja Figura 3.7. Para acessar um elemento, a implementação do método de acesso do *stxxl:vector* primeiramente verifica se a página à qual pertence o elemento requisitado está na *cache* do vetor. Se esse é o caso, a referência do elemento na cache é retornada. Caso contrário, a página é levada para a *cache*. Se não existir espaço livre na *cache*, então, uma página precisa ser levada de volta ao disco[2]. O vetor mantém um objeto *pager* que mostra qual página deverá ser retirada da *cache*. Essa página é selecionada de acordo com algumas estratégias implementadas pela STXXL tais como a LRU (*least recently used*) ou a estratégia de escolha aleatória. Além disso, a STXXL permite que o usuário defina sua própria estratégia. A estratégia mais eficiente e a *default* na STXXL é a LRU. Para cada página o vetor mantém um *flag* com o objetivo de rastrear se algum elemento na página foi modificado e, por conseguinte, precisa ser escrito no disco quando a página for retirada da *cache*. O *flag* é liberado sempre que a página for carregada na *cache*.

O stxxl :: vector possui uma série de parâmetros que precisam ser configurados no momento de sua criação, veja tabelas 3.1, 3.2 e 3.3. A Tabela 3.1 mostra os parâmetros do stxxl : vector, os valores default, os valores recomendados e os valores uti-

[1] Uma página é uma coleção de blocos consecutivos. O número de blocos na página é constante e dependente do Sistema Operacional.

[2] Se a página a ser levada de volta ao disco ainda não tiver sido acessada ou alterada, esse passo é pulado, pois a página não precisa ser escrita, pode ser descartada da *cache*. Para verificar essas situações, é utilizado um *flag* especial para cada página.

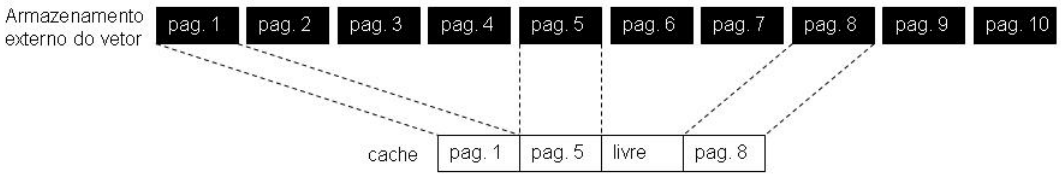


Figura 3.7: Esquema do *stxxl::vector*

lizados no algoritmo. Testes foram realizados com outros valores de parâmetros, mas os valores apresentados na Tabela 3.1 foram os que produziram melhores resultados no tempo de execução do algoritmo. Note que o único parâmetro que não recebeu o valor *default* foi o parâmetro *AllocaStr_*, que recebeu o valor *striping*, pois não foram utilizados discos em paralelo.

Parâmetro	Descrição	Valor		
		default	Recomendado	Utilizado
Tp_	tipo do elemento	-	-	ParOrdenado CelulaIndice
PgSz_	número de blocos em cada página	4	$\geq D$ (núm. de discos)	4
Pages_	número de páginas na cache	8	≥ 2	8
BlkSize_	tamanho do bloco em bytes	2097152 (2 MB)	quanto maior melhor	2097152 (2 MB)
AllocaStr_	estratégia de sinalização em discos paralelos (Tabela 3.2)	RC	RC	striping
Pager_	estratégia de paginação (Tabela 3.3)	lru	lru	lru

Tabela 3.1: Parâmetros de configuração do stxxl :: vector

No EMVS o stxxl :: vector é utilizado para representar dois tipos de lista: a lista de varredura que armazena pares (célula, índice) e a lista de pontos visíveis, que armazena apenas as coordenadas dos pontos visíveis. Pontos no EMVS são, na verdade, pares ordenados representados por dois valores inteiros indicando as coordenadas x e y do ponto na matriz de elevação. Já a célula é representada por um par ordenado e por um valor inteiro que indica a elevação do ponto na matriz, veja código 3.1. As listas no EMVS, então, são criadas com dois tipos: o ParOrdenado e a CelulaIndice, que armazena, além da célula, um inteiro indicando o índice da célula. Para facilitar a

identificação dos vetores na implementação do algoritmo, foram definidos identificadores para os dois tipos de vetor, veja abaixo:

#define VetorCell stxxl :: vector<CelulaIndice,4,stxxl :: lru_pager<8>,2097152,stxxl::striping>

#define VetorPar stxxl:: vector<ParOrdenado,4,stxxl::lru_pager<8>,2097152,stxxl::striping>

Código 3.1: Estruturas utilizadas na implementação do EMVS

```
#define tipoOrdem unsigned long long

struct ParOrdenado
{
    int x,y;
        bool operator==(const ParOrdenado &par2) const
        {
            return (this->x==par2.x && this->y==par2.y );
        }
};

struct CelulaIndice
{
  int x,y;
        short int elev;
        typedef tipoOrdem key_type;
        tipoOrdem ordem;
        CelulaIndice key() const { return ordem; };
        CelulaIndice() {}
        CelulaIndice(key_type __key):ordem(__key) {}
};
```

O stxxl :: vector possui uma série de membros. Aqueles utilizados pelo algoritmo EMVS estão listados na Tabela 3.4.

Com relação a esses membros, vale dizer que o stxxl :: vector possui o membro iterator além do const_iterator, porém o uso do iterator sem ser constante faz com que a página do elemento referenciado receba o flag de rastreamento citado anteriormente e, por consequência, precise ser levada de volta ao disco quando for retirada da *cache*, gerando operações de E/S desnecessárias. Por esse motivo, somente membros constantes são utilizados na implementação.O mesmo ocorre com o operador [.], sua versão sem ser constante existe, mas não a utilizamos.

Algoritmo de ordenação stxxl :: sort

O stxxl :: sort é um algoritmo de ordenação em memória externa equivalente ao std :: sort. O código 3.2 mostra o protótipo do método stxxl :: sort.

No protótipo, ExtIterator equivale a um iterador para estrutura em memória externa. No caso do EMVS, ExtIterator é substituído pelo membro const_iterator

Estratégia	Identificador
striping	striping
aleatório simples	SR
aleatório completo	FR
aleatório cíclico	RC

Tabela 3.2: Estratégias de sinalização em discos paralelos suportadas pela STXXL

Estratégia	Identificador
aleatório	random
least recently used	lru

Tabela 3.3: Estratégias de paginação suportadas pela STXXL

Membro	Descrição
const_iterator	Iterador constante usado para percorrer o vetor
const_iterator begin() const	Retorna um apontador para o iterador do primeiro elemento do vetor
const_iterator end() const	Retorna um apontador para o fim do vetor
syze_type size()	Retorna o tamanho do vetor
const_reference operator [ ] (size_type n) const	Retorna uma referência para o n-ésimo elemento do vetor
vector()	Cria um vetor vazio
void push_back(const T&)	Insere um novo elemento na última posição do vetor

Tabela 3.4: Membros do tipo stxxl:vector usados pelo EMVS

Código 3.2: Protótipo do método stxxl:sort

```
1 void sort (ExtIterator\_ inicio, ExtIterator\_ fim,
2             StrictWeakOrdering\_ cmp, unsigned M)
```

do stxxl::vector. Além disso, StrictWeakOrdering representa um modelo de ordenação que deve retornar, quando requisitado, valor mínimo e máximo para os elementos armazenados no vetor a ser ordenado. Na implementação do EMVS as estruturas CelulaIndiceCmp e ParOrdenadoCmp, mostradas no código 3.3, são usadas como StrictWeakOrdering e foram criadas para prover uma forma de comparação dos elementos do vetor durante a ordenação. Por fim, o parâmetro M limita a quantidade de memória interna que poderá ser usada pelo algoritmo de ordenação.

No EMVS, o stxxl::sort é utilizado duas vezes, uma para ordenar a lista de varredura, na qual a ordenação é realizada pela comparação de índices, e outra para ordenar a lista de pontos visíveis, onde a ordenação é realizada pela comparação das coordenadas dos pontos, veja linha 23 no Código 3.4 e linha 37 no Código 3.7.

Código 3.3: Estruturas utilizadas para comparar elementos do tipo CelulaIndice e ParOrdenado pelo método stxxl::sort

```
1  struct CelulaIndiceCmp
2  {
3    bool operator () (const cell & a, const cell & b) const
4    {
5      return a.ordem < b.ordem;
6    }
7
8    static cell min_value()
9    {
10     return CelulaIndice(std::numeric_limits<tipoOrdem>::min());
11   }
12
13   static cell max_value()
14   {
15     return CelulaIndice(std::numeric_limits<tipoOrdem>::max());
16   }
17
18 };
19
20 struct ParOrdenadoCmp
21 {
22   bool operator() (const parOrdenadoInt & a,
23                    const parOrdenadoInt & b) const
24   {
25     return ( a.x + nrows*a.y < b.x + nrows*b.y );
26     //nrows é o número de linhas da matriz de elevação
27   }
28
29   static ParOrdenado min_value()
30   {
31     ParOrdenado c;
32     c.x = -1;
33     c.y = -1;
```

```
34      return c;
35    }
36
37    static ParOrdenado max_value()
38    {
39      ParOrdenado c;
40      c.x = std::numeric_limits<int>::max();
41      c.y = std::numeric_limits<int>::max();
42      return c;
43    }
44  };
```

Resumidamente, o stxxl::sort ordena os elementos no intervalo [inicio,fim) em uma ordem ascendente definida pelo parâmetro cmp. Isso significa que se i e j são dois iteradores válidos em [inicio,fim) tal se i precede j, então, na estrutura ordenada, i precederá j.

Geração da lista de varredura

A geração da lista de varredura é realizada pela função apresentada no Código 3.4. Essa função recebe como parâmetros o número de linhas da matriz de elevação (nrows), as coordenadas de representação do observador (ox, oy) bem como sua altura (observerElev), o raio de interesse (roi) e o tamanho ocupado por cada elevação na matriz, em bytes, (elsize), e retorna a lista de varredura do tipo VetorCell.

Essa função define inicialmente as coordenadas (x_0, y_0) e (x_1, y_1) que limitam o quadrado de varredura. Então, a função percorre cada célula do quadrado de varredura e faz uma chamada a outra função (determinaIndicesCelula) que determina todos os índices da célula e insere o par (célula, índice) na lista de varredura (listaVarredura). As linhas 13 a 22 do Código 3.4 mostram o algoritmo percorrendo o quadrado de varredura. Antes de determinar os índices de uma célula, a função verifica se ela faz parte do terreno e do círculo determinado pelo raio de visão, através da função pertenceAoCirculo.

Note que a matriz de elevação é lida da entrada padrão *cin* e que a variável elsize é usada para indicar o número de bytes a serem lidos do arquivo contendo a matriz de elevação. Ao final, a função retorna a lista de varredura já ordenada pelos índices das células.

Código 3.4: Geração da lista de varredura

```
1  VetorCell* geraListaVarredura(int nrows, int ox, int oy,
2                                int observerElev, int roi, int elsize)
3  {
4    short int elev;
```

```
5   VetorCell  *listaVarredura;
6   listaVarredura = new VetorCell;
7   assert(streamSaida!=NULL);
8   int x0 = (ox-roi) >=0 ? (ox-roi):0;
9   int y0 = (oy-roi) >=0 ? (oy-roi):0;
10  int x1 = (ox+roi) >=nrows ? (nrows-1): ox+roi;
11  int y1 = (oy+roi) >=nrows ? (nrows-1): oy+roi;
12
13  for (int y=y0;y<=y1;y++)
14  {
15    cin.seekg( 2*(y*nrows + x0),ios::beg);
16    for (int x=x0;x<=x1;x++)
17    {
18      cin.read(reinterpret_cast<char*>(&elev), elsize);
19      if (pertenceAoCirculo(x,y,ox,oy,roi))
20        determinaIndicesCelula(x,y,ox,oy,elev,roi,*listaVarredura);
21    }
22  }
23  stxxl::sort(listaVarredura->begin(),listaVarredura->end(),
24              CelulaIndiceCmp(),780*1024*1024);
25  return listaVarredura;
26 }
```

Determinação dos índices das células

A determinação de todos os índices de uma célula é a parte principal do algoritmo EMVS. A função determinaIndicesCelula recebe como parâmetro as coordenadas e a elevação da célula em questão (x,y,elev), as coordenadas do observador (ox,oy), o raio de visão (roi) e a lista onde serão armazenados todos os pares (célula, índice) determinados (listaVarredura), veja o Código 3.5.

Código 3.5: Determinação dos índices de uma célula

```
1  void determinaIndicesCelula(int x, int y, int ox, int oy,
2
3                              VetorCell &listaVarredura)
4  {
5
6    ParOrdenado celulaTocada1,celulaTocada2;
7    parOrdenadoDouble ondeToca1,ondeToca2; //parOrdenadoDouble é uma
8                                           //estrutura que possui duas
9                                           //coordenadas double x e y
10         if (x==ox && y==oy)
11         {
12           return;
13         }
14         // Para facilitar os cálculos abaixo, consideraremos um eixo
15         //cartesiano com o observador em (0,0)
16         x-=ox;
17         y-=oy;
```

```
18
19    ondeToca1 = ondeToca( (x>=y) ? (x):(x+0.5) ,
20                          (x>=y)?(y-0.5):(y) ,roi);
21    ondeToca2 = ondeToca( (x>y ) ? (x):(x-0.5) ,
22                          (x>y) ?(y+0.5):(y) ,roi);
23    celulaTocada1.x = (int) floor(ondeToca1.x);
24    celulaTocada1.y = (int) ceil(ondeToca1.y);
25    celulaTocada2.x = (int) ceil(ondeToca2.x);
26    celulaTocada2.y = (int) floor(ondeToca2.y);
27
28    int los1 = nEsimaVarredura(celulaTocada1.x,
29                                    celulaTocada1.y,roi);
30    int los2 = nEsimaVarredura(celulaTocada2.x,
31                                    celulaTocada2.y,roi);
32    int numLos =  los2 - los1 + 1;
33    int nEsimaLos = los1;
34
35    int mx = abs(x);
36    int my = abs(y);
37    CelulaIndice temp;
38    for (int contador=0;contador<numLos;contador++)
39    {
40      temp.x = x;
41      temp.y = y;
42      temp.elev = elev;
43      temp.ordem =(tipoOrdem)(((tipoOrdem)nEsimaLos+contador)*
44                              ((tipoOrdem)roi*RAIOMULT)+
45                              ((mx>my)?mx:my) );
46      listaVarredura.push_back(temp) ;
47    }
48  }
```

Vamos chamar de c a célula em questão processada pela função. Então, a função inicialmente determina as duas células da borda do quadrado de varredura que fazem parte da primeira e da última LOS (em sentido anti-horário) que tocam a célula c. No Código 3.5 a variável celulaTocada1 representa a célula que faz parte da primeira LOS e a variável celulaTocada2 representa a célula que faz parte da última LOS. Vale ressaltar que, para simplificar a explicação, os valores para cálculo das células celulaTocada1 e celulaTocada2 consideram que c pertence ao primeiro quadrante da matriz de elevação, para os demais quadrantes cálculos semelhantes são realizados.

A função ondeToca usada nas linhas 9 e 11 do Código 3.5 usa o coeficiente angular da reta que passa por c e pelo observador para determinar a célula da borda do quadrado de varredura que faz parte da LOS que toca c, veja no Código 3.6.

Código 3.6: Determinação da célula da borda do quadrado de varredura que passa por uma determinada LOS

```
1  inline parOrdenadoDouble ondeToca(double x, double y, int raio) {
```

```
2
3   parOrdenadoDouble celula; //variável que possui duas coordenadas
4                             //double x e y
5   if ( abs(x) > abs(y) )
6   {
7     double coeficienteAngular = abs(y)/abs(x);
8     double yTocado = coeficienteAngular*raio;
9     celula.y= (y>0) ? yTocado:-yTocado;
10    celula.x=(x>0)?raio:-raio;
11  }
12  else
13  {
14    double coeficienteAngular2 = abs(x)/abs(y);
15    double xTocado = coeficienteAngular2*raio;
16    celula.x=(x>0)?xTocado:-xTocado;
17    celula.y=(y>0)?raio:-raio;
18  }
19  return celula;
20 }
```

Após a determinação de celulaTocada1 e celulaTocada2 o número da LOS que passa em cada uma dessas células é calculado, veja as linhas 28 e 30 do Código 3.5. A função nEsimaVarredura realiza essa atividade atribuindo valor às variáveis los1 e los2, devido a sua simplicidade, a função nEsimaVarredura não será apresentada. Daí, para cada uma das linhas de visada no intervalo [los1, los2), o índice da célula correspondente à linha é determinado na linha 43 através da equação (3.5) [3]. Logo após, o par (célula, índice) é inserido na lista de varredura (listaVarredura), veja linha 46 do Código 3.5.

Cálculo do *viewshed*

Uma vez que possuímos todas as células com seus índices inseridos na lista de varredura, o cálculo da visibilidade de cada uma das células é efetuado. A função calculaViewshed realiza esse cálculo recebendo como parâmetros a lista de varredura (listaVarredura), já ordenada pelos índices das células, a lista de pontos visíveis (listaPontosVisiveis) vazia , as coordenadas do observador (ox, oy) bem como sua elevação (observerElev), além da altura do alvo que deve ser visto pelo observdaor (targetHt). Para cada célula c da lista de varredura, a função calculaViewshed determina a inclinação da reta que liga o observador à célula c, mostrada na linha 22, através da equação (3.1). Se essa inclinação s é maior do que a armazenada até o momento em horizon_slope, então a variável horizon_slope é atualizada com o valor de s.

[3] Usamos na equação a constante RAIOMULT, de valor 100, para separar melhor os índices das células que pertencem à linhas de visada diferentes

Código 3.7: Determinação da visibilidade da célula

```
1  void calculaViewshed(VetorCell* listaVarredura,
2                       VetorPar* listaPontosVisiveis,
3                      int ox, int oy, int observerElev, int targetHt)
4  {
5    VetorCell::const_iterator listaIterator=listaVarredura->begin();
6    CelulaIndice *celula = new CelulaIndice();
7    CelulaIndice c;
8    int pElev=0;
9    double horizon_slope, slope, s, horizon_alt;
10   bool v;
11   int inciny;
12
13   for(;listaIterator!=listaVarredura->end();listaIterator++)
14   {
15     c = (*listaIterator);
16     celula = &c;
17     pElev = listaIterator->elev;
18
19     inciny = abs (celula->x - ox) /
20              abs (celula->y - oy);
21
22     s =(double) (pElev - observerElev) / (double) abs
23                 (inciny?celula->y:celula->x -
24                 inciny?oy:ox);
25     if (horizon_slope < s)       horizon_slope = s;
26
27     horizon_alt =        observerElev + horizon_slope *
28                   abs (inciny?celula->y:celula->x -
29                   inciny?oy:ox);
30     v = (pElev + targetHt >= horizon_alt);
31
32     if (v)
33     {
34       listaPontosVisiveis->push_back( celula );
35     }
36   }
37   stxxl::sort(listaPontosVisiveis->begin(),listaPontosVisiveis->end(),
38               ParOrdenadoCmp(),(memMax-20)*1024*1024);
39 }
```

Para determinar se a célula é visível ou não, um outro cálculo é realizado envolvendo as alturas do observador (observerElev) e do alvo (TargetHt) que será posicionado na célula e visto pelo observador. Esse cálculo tem como objetivo identificar situações em que o ponto onde será posicionado o observador não enxerga o ponto onde será posicionado o alvo, mas devido a altura do alvo ser maior que a altura do observador, esse consegue enxergar aquele e, portanto, a célula pode ser definida como visível. Esse cálculo pode ser visto nas linhas 27 a 30 do Código 3.7. Na implementação do EMVS, utilizamos como altura para o alvo o mesmo valor da elevação do observador, porém esse dado pode ser configurado para ser uma entrada do usuário.

Toda célula visível é inserida na lista de pontos visíveis listaPontosVisiveis, que é ordenada antes do fim da função.

Geração da matriz de *viewshed*

A matriz de *viewshed* é gerada através da consulta à lista de pontos visíveis (visibleStream) após a ordenação desta utilizando as coordenadas dos pontos como chave de ordenação. A função geraMatrizViewshed utiliza dois laços para iterar entre as coordenadas x e y dos pontos da matriz de elevação, veja Código 3.8. A partir daí, para cada ponto da matriz a função verifica se as coordenadas são as mesmas do ponto referenciado pelo iterador da lista de pontos visíveis. Se as coordenadas forem iguais então a célula é visível e o iterador é incrementado, caso contrário, a célula não é visível.

Código 3.8: Geração da matriz de *viewshed*

```
1  void geraMatrizViewshed(VetorPar *visibleStream) {
2    unsigned char mask[8] = { 128, 64, 32, 16, 8, 4, 2, 1 };
3    char tempGravar = 0;
4    int i=0;
5
6    VetorPar::const_iterator visibleIterator=visibleStream->begin();
7
8    long long num_bytes = (((long long)nrows)*nrows+7)/8;
9
10   for (int y=0;y<nrows;y++)
11     for (int x=0;x<nrows;x++) {
12
13       if (i==8)
14       {
15         i=0;
16         cout.write(&tempGravar,1);
17         tempGravar = 0;
18       }
19       if (x == visibleIterator->x && y== visibleIterator->y ) {
20         tempGravar |= mask[i];
21         visibleIterator++;
22         if (visibleIterator != visibleStream->end() )
23         {
24           while (x == visibleIterator->x && y ==visibleIterator->y) {
25             if (visibleIterator == visibleStream->end() ) {
26               cout.write(&tempGravar,1);
27               long long numBytesGravados = (y*nrows+x+7)/8;
28               long long numRestantes = nb-numBytesGravados;
29               tempGravar=0;
30               for (;numRestantes>0;numRestantes--)
31                 cout.write(&tempGravar,1);
32               cout << flush ;
```

```
33                  return;
34                }else  {
35                  visibleIterator++;
36                }
37              }
38            } else {
39              cout.write(&tempGravar,1);
40              long long numBytesGravados = (((long long)y)*nrows+x+7)/8;
41              long long numRestantes = nb-numBytesGravados;
42              tempGravar=0;
43              for(;numRestantes>0;numRestantes--)
44                cout.write(&tempGravar,1);
45                cout << flush;
46                return;
47            }
48          }
49          i++;
50        }
51    cout.write(&tempGravar,1);
52    cout << flush;
53    return;
54  }
```

A escrita na matriz do *viewshed* é realizada de forma otimizada e será explicada na próxima seção.

Otimização do Algoritmo

Para melhorar o desempenho do algoritmo EMVS eliminando operações de E/S, algumas otimizações foram realizadas em sua implementação. A primeira otimização realizada consiste em manter uma parte da matriz de elevação em memória interna. Mais especificamente, o algoritmo mantém a maior região possível ao redor do observador na memória interna. Dessa forma, quando uma célula precisa ser processada, o algoritmo verifica se ela faz parte da região mantida em memória e somente em caso negativo, acessa o disco para obter a célula. A região ao redor do observador é a escolhida pois as células dessa região, como mostrado na seção 4.1, são interceptadas por diversas linhas de visada e, portanto, são as células que são processadas o maior número de vezes durante a execução do programa.

Outra otimização se refere ao processo de escrita no arquivo que armazena a matriz do *viewshed*. Visto que a visibilidade de cada célula é indicada por um bit, então o processo de escrita agrupa as células em blocos com 8 células correspondente a um byte, evitando que a cada bit seja realizada uma operação de escrita no disco.

3.2.2 Complexidade

O projeto do algoritmo EMVS utiliza o PDM (Vitter e Shriver, 1994) como modelo de memória assumindo a comunicação da memória RAM com apenas um disco magnético. As principais medidas de desempenho para algoritmos e estruturas de dados no PDM são, em ordem de relevância: o número de operações de E/S realizadas; a quantidade de espaço em disco utilizado; o tempo de processamento interno (sequencial ou paralelo). Porém, a grande maioria dos algoritmos projetados sob este modelo se concentra nas duas primeiras medidas, considerando o tempo de processamento interno livre.

Com relação à segunda medida, algoritmos e estruturas de dados ideais devem usar espaço de memória linear, ou seja, $O(N/B)$ blocos do disco para o armazenamento total, veja capítulo 2 para maiores explicações sobre os parâmetros N e B. Já para a primeira medida de desempenho, o número de operações de E/S realizadas por algoritmos e estruturas de dados em memória externa geralmente é expresso em função dos limites das operações $scan(N)$ e $sort(N)$. Para a demonstração de complexidade do EMVS utilizaremos apenas o número de operações de E/S, que é a medida decisiva para os algoritmos em memória externa.

Seja T o terreno representado por uma matriz de elevação de dimensões $n \times n$. Além disso, seja p a posição do observador e seja r o raio de interesse. O primeiro passo do algoritmo EMVS é determinar a ordem de processamento das células, criando a lista de varredura L e, para isso, precisa ler todas as células do disco, executando $\frac{n^2}{B}$ operações de E/S.

Após a leitura inicial da matriz de elevação, o algoritmo processa apenas as células que estão dentro do quadrado de dimensões $2r \times 2r$ centrado em p. Assumindo que o lado de cada célula é s, então, haverá, no máximo, $\frac{2r}{s}$ células em cada lado do quadrado o que implica em $\frac{8r}{s}$ células no perímetro do quadrado. Seja $K = \frac{r}{s}$. Como o algoritmo traça as linhas de visada, utilizadas na varredura radial, partindo do observador p e passando por cada uma das células da borda do quadrado de varredura, então, o algoritmo traça $8K$ linhas de visada. Cada linha de visada é determinada através da rasterização de linhas, e portanto possui no máximo K células. A lista L, então, possui, no pior caso, $O(K^2)$ elementos.

Em um segundo passo, a lista L é ordenada e precisa ser varrida para a realização do cálculo de visibilidade das células, o que gera $O(sort(K^2))$ operações de E/S para a ordenação e $O(scan(K^2))$ operações de E/S para a varredura. O algoritmo ainda usa outra lista L', a lista de pontos visíveis, que precisa ser ordenada e também percorrida para gerar o arquivo final de visibilidade (matriz de *viewshed*). Supondo

que o número de elementos em L' seja $O(Z)$, as operações de ordenação e varredura em L' geram mais $O(sort(Z))$ e $O(scan(Z))$ operações de E/S.

Assim, o número total de operações de E/S no EMVS é dado por:

$$O\left(\frac{n^2}{B}\right) + O\left(\frac{K^2}{B}\log_{\left(\frac{M}{B}\right)}\left(\frac{K^2}{B}\right)\right) + O\left(\frac{K^2}{B}\right) + O\left(\frac{Z}{B}\log_{\left(\frac{M}{B}\right)}\left(\frac{Z}{B}\right)\right) + O\left(\frac{Z}{B}\right)$$

Normalmente, o raio de interesse r é (muito) menor do que n (o lado da matriz do terreno), e além disso geralmente o tamanho da lista L' é (muito) menor do que o tamanho de L. Assim, K é menor do que n e Z é menor que K^2, então, o número de operações de E/S é dada por $O(\frac{n^2}{B}) = O(scan(n^2))$.

O pior caso (não usual) do algoritmo ocorre se o raio de interesse é grande a ponto de cobrir praticamente todo o terreno, nesse caso, o número de operações de E/S é $O\left(\frac{K^2}{B}\log_{\left(\frac{M}{B}\right)}\left(\frac{K^2}{B}\right)\right) = O(sort(K^2))$, onde $K = \frac{n}{s}$.

Na implementação do algoritmo, um ganho de eficiência é alcançado armazenando-se uma parte da matriz do terreno em memória interna. Nesse caso, as células que estão em memória interna não são inseridas em L e L' e, quando uma célula precisa ser processada, o algoritmo verifica se essa célula está em memória interna. Se estiver, ela é processada normalmente; caso contrário, ela é lida de L.

Supondo que o número de células que podem ser armazenadas na memória interna seja dado por M, então o número de operações de E/S do EMVS com a otimização realizada é dado por:

$$O\left(\frac{n^2}{B}\right) + O\left(\phi\log_{\left(\frac{M}{B}\right)}\phi\right) + O\left(\phi\right) + O\left(\sigma\log_{\left(\frac{M}{B}\right)}\sigma\right) + O\left(\sigma\right)$$

onde $\phi = \frac{K^2-M}{B}$ e $\sigma = \frac{Z-M}{B}$.

Conforme dito anteriormente, K é muito menor que n, assim como Z é menor que K^2. Além disso, M é, geralmente, muito menor que n quando lidamos com grandes terrenos e, então, temos que o número de operações de E/S é também $O(scan(n^2))$. No pior caso, o número de operações de E/S se dá por $O\left(\frac{K^2-M}{B}\log_{\left(\frac{M}{B}\right)}\left(\frac{K^2-M}{B}\right)\right) = O(sort(K^2-M))$, onde $K = \frac{n}{s}$.

Capítulo 4

Avaliação de Eficiência

No capítulo anterior o EMVS foi apresentado em conjunto com sua implementação e complexidade. Esse capítulo apresenta os testes aplicados ao algoritmo e seus resultados mostrando na prática o desempenho do algoritmo e atestando sua eficiência.

Para obter o tempo de execução do algoritmo, o mesmo foi aplicado a terrenos de diferentes tamanhos, considerando diferentes raios de interesse. E para avaliar seu desempenho, o tempo de execução foi comparado com o de dois outros algoritmos com objetivos distintos. A comparação com o algoritmo de Franklin e Ray foi realizada para demonstrar a aplicabilidade do EMVS em terrenos maiores que a memória interna e com o algoritmo de Haverkort et al. (Haverkort et al., 2007) para comprovar o ganho de eficiência obtido pelo uso do EMVS.

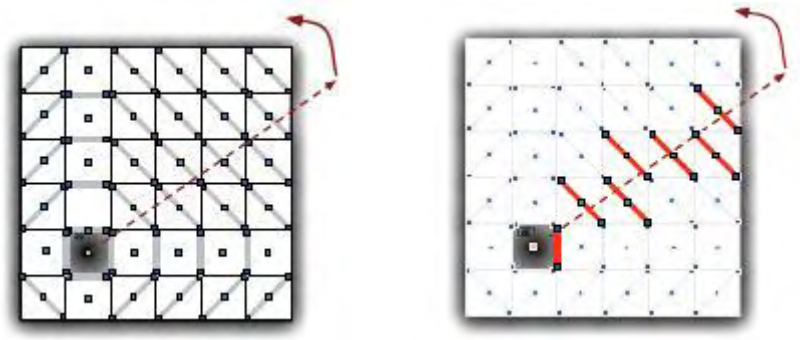
As seções abaixo descrevem o algoritmo de Haverkort et al. e apresentam a avaliação da eficiência do EMVS.

4.1 O algoritmo de Haverkort et al.

O algoritmo de Haverkort et al. (Haverkort et al., 2007), da mesma forma que o EMVS, considera o problema de computar em memória externa o *viewshed* de MDEs muito grandes. Esse algoritmo é uma adaptação do método proposto por Van Kreveld (Van Kreveld, 1996) que será brevemente descrito a seguir.

Dado um terreno representado por uma matriz de elevação T de dimensões $n \times n$ e dado um ponto p em T o algoritmo de Van Kreveld utiliza a técnica de varredura no plano ou *plane sweep* para calcular o *viewshed* do observador p. A idéia básica consiste em rotacionar uma linha, chamada de linha de varredura, ao redor de p e determinar a visibilidade de cada célula da matriz quando a linha passar sobre seu centro. Para que isto seja feito, o algoritmo mantém uma estrutura de dados,

chamada de estrutura de células ativas, que, durante todo o processo, armazena as células interceptadas pela linha de varredura ou células ativas, veja Figura 4.1(b). Uma célula é inserida nessa estrutura quando é interceptada pela linha de varredura e retirada da estrutura quando a linha finaliza essa interseção. Além disso, quando o centro de uma célula é interceptado, a estrutura de células ativas é consultada para determinar se tal célula é ou não visível por p. Dado todo esse processo, a célula pode ser associada a três eventos: quando é interceptada pela linha pela primeira vez e é inserida na estrutura; quando a linha passa sobre seu centro; e quando deixa de ser interceptada pela linha e é retirada da estrutura.



(a) (b)

Figura 4.1: Algoritmo de varredura no plando de Van Kreveld: (a) Cada célula é marcada com seus três eventos; (b) Células ativas

O algoritmo também utiliza a idéia da LOS para determinar se uma célula é visível. De acordo com o processo descrito acima, uma célula é dita visível se, no momento em que a linha de varredura passar por seu centro, não existir nenhuma célula ativa mais próxima a p e com maior altura acima do horizonte. Dada uma célula Q na matriz, seja q seu centro, então, a altura de Q acima do horizonte com relação ao observador p é definida como a altura de q acima do horizonte, que é dada por

$$altura(q) = \arctan \frac{e_q - e_p}{d_{pq}}$$

onde e_q e e_p são as elevações de q e p, respectivamente, e d_{pq} é a distância entre p e q. Nesse caso, a célula inteira é tratada como se tivesse altura constante com relação a

p.

Para consultar de forma eficiente as células ativas, o algoritmo mantém essas células numa árvore binária balanceada, onde as células ativas são armazenadas da esquerda para a direita em ordem crescente de distância entre o centro da célula e observador. Além disso, cada nó armazena a maior altura acima do horizonte na subárvore que possui esse nó como raiz; as folhas armazenam a altura acima do horizonte da própria célula contida nelas.

Para adaptar o algoritmo de Van Kreveld para a memória externa, Haverkort et al. consideram quatro estruturas: uma matriz de elevação, uma matriz de visibilidade, uma lista de eventos e uma estrutura ativa. Para alcançar a complexidade de $O(sort(n^2))$ os autores combinam as técnicas de varredura e distribuição, utilizando a técnica de varredura por distribuição. Maiores detalhes do algoritmo podem ser encontrados no Apêndice A.

4.2 Experimentos e avaliação de eficiência

Essa seção apresenta os experimentos realizados com o EMVS e sua comparação com o algoritmo de Franklin e Ray (WRF_VS) e com o algoritmo de Haverkort et al. (IO_VS).

4.2.1 Resultados experimentais

O algoritmo EMVS foi implementado em C++, usando g++ 4.1.1, e para alcançar a eficiência em operações de entrada e saída o algoritmo utiliza a *STXXL*, também em C++. Todos os experimentos foram realizados em um PC Pentium 2.8 GHz, 1 GB de RAM, HD Serial ATA de 80 GB e 7200 RPM e sistema operacional Mandriva Linux. Para uma melhor avaliação do custo das operações de E/S, foram consideradas duas configurações para a plataforma: uma utilizando o total da RAM de 1 GB e permitindo ao programa o uso de 800 MB para dados e uma outra utilizando somente 256 MB e permitindo o uso de 200 MB de dados. Apesar de uma memória interna de 256 MB ser muito pequena para PCs atuais, nós utilizamos esse tamanho por duas razões: para ilustrar o comportamento do algoritmo quando a diferença entre o tamanho do terreno e o tamanho da memória interna aumenta e, também, porque isso nos dá uma idéia do desempenho do algoritmo em dispositivos portáteis onde há pouca memória e pouco espaço no disco.

O terreno usado nos testes foi obtido na página do NASA SRTM (NASA's Shuttle Radar Topography Mission (SRTM), 2009) e representa a América

do Sul (especialmente o Brasil), veja Figura 4.2(a). Os dados dos terrenos foram amostrados a uma resolução de 3 segundos (aproximadamente 90m). Vale ressaltar que essas regiões contêm uma porcentagem muito pequena (menos que 1,5%) de dados inválidos (isto é, pontos onde a elevação é desconhecida ou inválida) e, portanto, os resultados não são influenciados por esses dados.

Para realização dos testes, foram selecionadas algumas porções do território do Brasil com diferentes tamanhos, veja Figura 4.2(b) Para cada tamanho de terreno, foi determinada a média do tempo necessário para processar os dados do terreno situando o observador em cinco posições diferentes, escolhidas aleatoriamente entre as posições do interior do quadrado definido pelas coordenadas ($n/2$ - $r/5$,$n/2$ - $r/5$) e ($n/2$ + $r/5$,$n/2$ + $r/5$), onde n é o tamanho do lado do terreno e r é o raio de interesse. Os valores dos limites do quadrado de posicionamento do observador foram determinados de forma a manter o observador no centro do terreno.

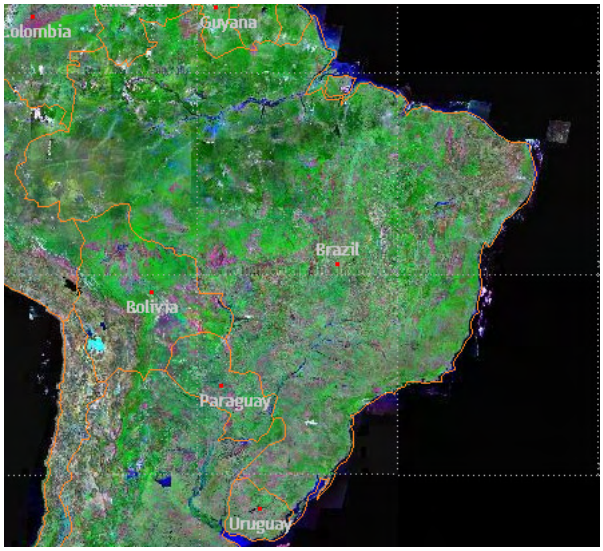
A Tabela 4.1 mostra o tempo de execução do EMVS em terrenos usando 1GB e 256MB de RAM. Nós sempre consideramos o pior caso, ou seja, usamos um raio de interesse grande o bastante para cobrir todo o terreno. O tempo gasto com o "processamento externo" (isto é, operações de E/S, ordenação no disco rígido, acesso a arquivo, etc) é mostrado separadamente na coluna *Ext.* e o tempo total de execução é mostrado na coluna *Tot.*. Além disso, para avaliar a influência do número de pontos visíveis (a coluna *# Pts Vis.*) no tempo de processamento, o observador foi posicionado a diferentes alturas (1, 50, 100, 1000, e 10000 metros) acima do terreno. Embora 1000 e 10000 metros não sejam elevações muito comuns em aplicações práticas, nós apresentamos esses resultados para confirmar a escalabilidade do algoritmo EMVS.

As figuras 4.3, 4.4, 4.5 e 4.6 resumem os tempos médios de processamento interno e externo. Como esperado, o tempo de processamento externo é muito maior em terrenos maiores do que a memória interna; veja os gráficos 4.5, 4.6 e, principalmente, 4.4, onde o tempo de processamento externo é maior que o de processamento interno quando se usa 256 MB e menor que se usa 1 GB. Essa diferença em 4.4 ocorre porque o terreno de 1122 MB pode ser armazenado quase que completamente na memória de 1 GB, necessitando, assim, de menos operações de E/S. Por outro lado, quando usamos 256 MB é necessário um número maior de operações de E/S.

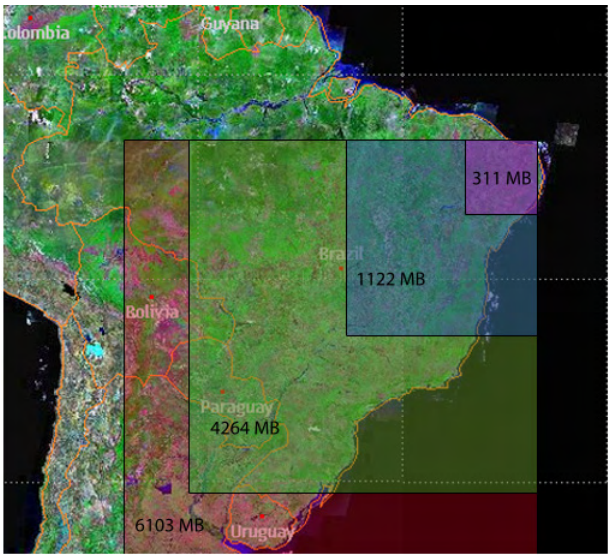
Note que, quando o tamanho do terreno aumenta, o tempo de processamento total é essencialmente determinado pelo tempo de processamento externo, que parece convergir para um valor aproximado de 80% e 75% do tempo total, utilizando 256 MB e 1 GB, respectivamente.

Tamanho Terreno	Altura Obs.	# Pts Vis.	Tempo (s) 1 GB Externo	Tempo (s) 1 GB Total	Tempo (s) 256 MB Externo	Tempo (s) 256 MB Total
9029^2 311 MB	1	3.3×10^5	16	40	15	40
	50	8.5×10^6	15	43	15	44
	10^2	1.9×10^7	16	47	15	47
	10^3	6.2×10^7	16	62	16	62
	10^4	6.2×10^7	15	62	16	63
17150^2 1122 MB	1	2.9×10^5	57	186	379	549
	50	7.5×10^6	57	192	372	548
	10^2	2.2×10^7	58	199	373	555
	10^3	2.2×10^8	58	274	373	634
	10^4	2.2×10^8	58	276	374	637
33433^2 4264 MB	1	2.4×10^4	1521	2309	3256	4133
	50	5.6×10^5	1563	2309	3809	4693
	10^2	4.6×10^6	1514	2308	3514	4400
	10^3	7.4×10^8	1498	2600	5292	6511
	10^4	8.5×10^8	1499	2632	5427	6685
40000^2 6103 MB	1	1.6×10^6	3843	5055	6154	7504
	50	7.3×10^7	4282	5548	6869	8263
	10^2	1.3×10^8	4592	5908	4592	8388
	10^3	1.1×10^9	4997	6722	8259	10094
	10^4	1.2×10^9	5008	6749	8640	10529

Tabela 4.1: Tempo médio de execução do EMVS (em segundos) utilizando 1 GB e 256 MB em porções do terreno do Brasil com diferentes tamanhos e diferentes posicionamentos do observador e variando a altura do observador acima do terreno (gerando *viewshed* com número diferente de pontos visíveis - mostrado na coluna # Pts Vis.). Em todos os casos, o raio de interesse cobre todo o terreno.



(a)



(b)

Figura 4.2: Imagem de satélite do terreno usado para os testes: (a) visão geral do terreno e (b) separação das regiões de diferentes tamanhos

4.2.2 Comparação com Franklin e Ray

O algoritmo de Franklin e Ray, identificado nesta seção por WRF_VS, foi projetado de modo a obter bom desempenho no cálculo do *viewshed* de terrenos menores que a memória interna, possuindo limitações quando é necessária a realização de operações

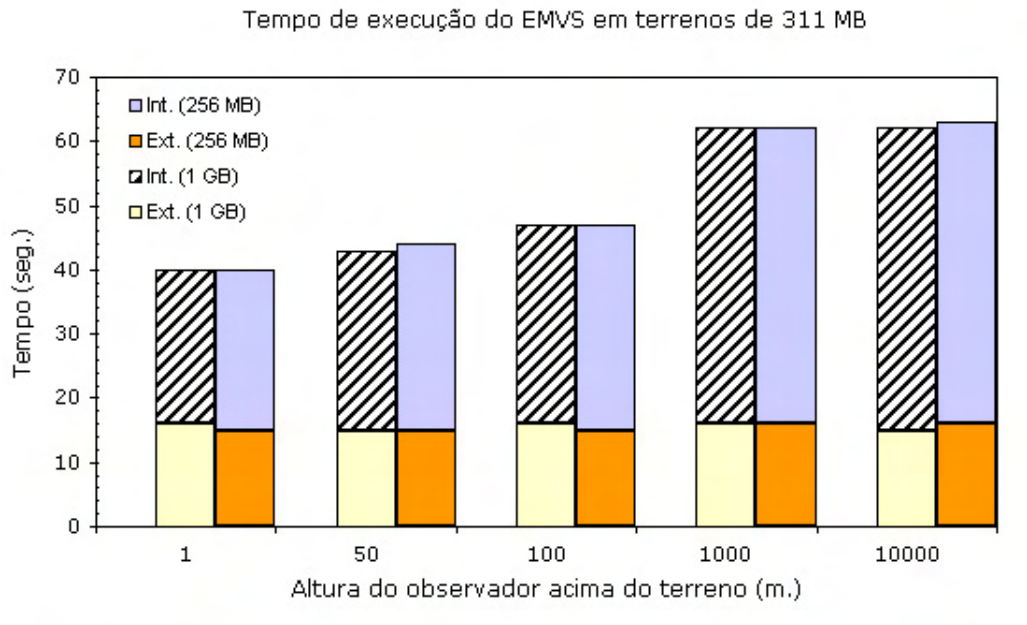


Figura 4.3: O tempo de processamento interno e externo usando 256 MB e 1 GB de RAM em terrenos de 311 MB.

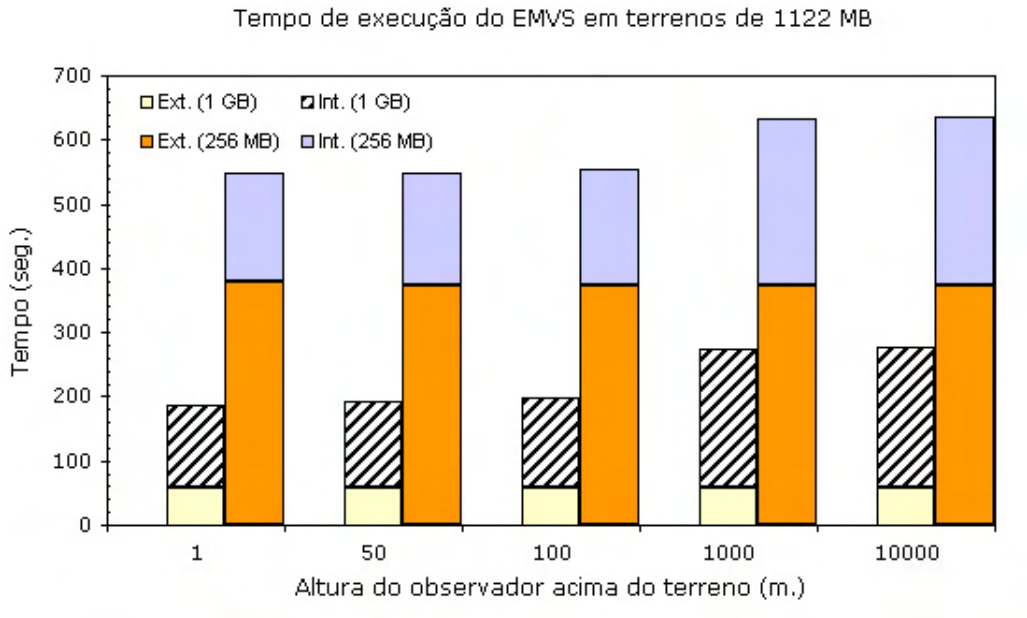


Figura 4.4: O tempo de processamento interno e externo usando 256 MB e 1 GB de RAM em terrenos de 1122 MB.

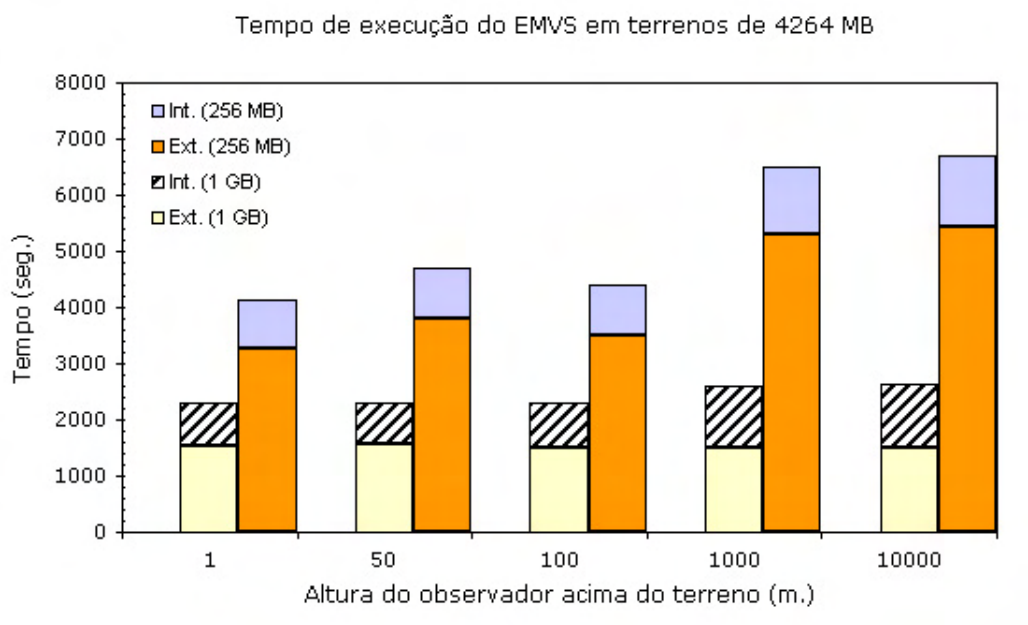


Figura 4.5: O tempo de processamento interno e externo usando 256 MB e 1 GB de RAM em terrenos de 4264 MB.

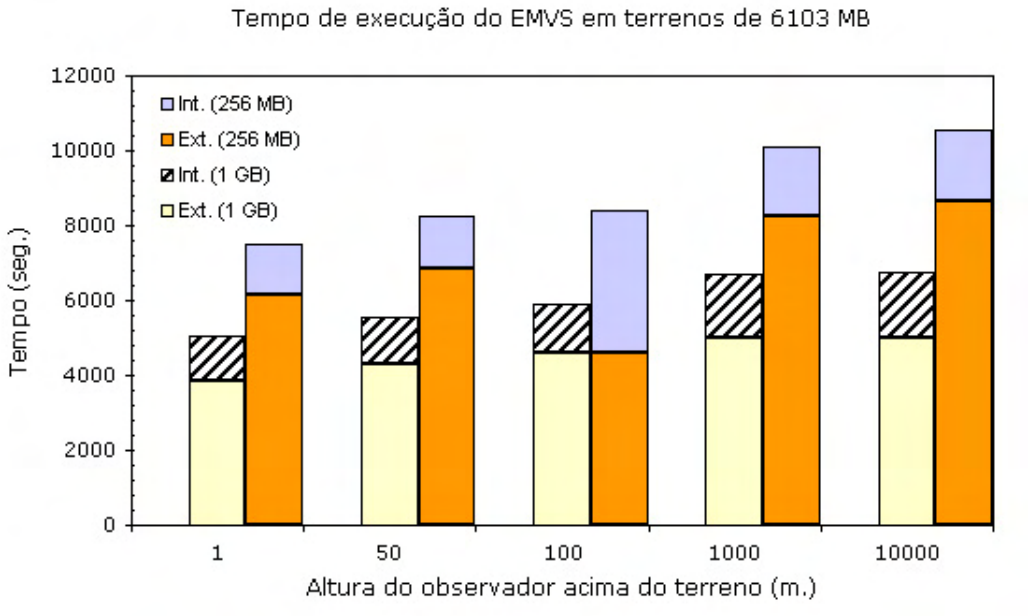


Figura 4.6: O tempo de processamento interno e externo usando 256 MB e 1 GB de RAM em terrenos de 6103 MB.

de E/S. Para que os experimentos realizados no EMVS pudessem ser comparados com o algoritmo de Franklin e Ray de forma a comprovar a melhoria causada por ser uso, efetuamos uma pequena modificação nesse algoritmo para que fosse possível processar terrenos maiores que a memória interna.

A modificação do algoritmo de Franklin e Ray consistiu em permitir que os dados fossem acessados diretamente na memória externa. Por questões de eficiência, parte da matriz de elevação é mantida na memória interna. Dessa forma, quando uma célula precisa ser processada, o algoritmo verifica se ela faz parte da região mantida em memória, e somente em caso negativo acessa o disco para obter a célula. Essa modificação elimina uma série de acessos ao disco e possibilita que o algoritmo manipule terrenos maiores do que sua versão original.

Para gerar os resultados, aplicamos o algoritmo de Franklin e Ray e o EMVS em três terrenos e calculamos a média do tempo necessário para processar esses terrenos com diferentes raios de interesse e situando o observador em três posições aleatoriamente escolhidas entre as posições do quadrado definido pelas coordenadas ($n/2$ - $r/5$,$n/2$ - $r/5$) e ($n/2$ + $r/5$,$n/2$ + $r/5$), onde n é o tamanho do lado do terreno e r é o raio de interesse. A Tabela 4.2 resume esses resultados considerando apenas a configuração de memória com 1 GB, pois com uma memória de 256 MB o algoritmo de Franklin e Ray, mesmo modificado, não consegue processar terrenos muito grandes. Os terrenos de 1.45 GB e 5.7 GB foram gerados à partir da concatenação de 4 e 16 instâncias, respectivamente, de uma matriz de 363 MB representando o Havaí, veja Figura 4.7. Já o terreno de 2GB foi gerado pela concatenação de várias instâncias da matriz de 5,5 MB representando o lago *Champlain West* situado na divisa entre os Estados Unidos e o Canadá, veja Figura 4.8. Esses terrenos são interessantes para esses testes, pois possuem grandes variações de altitudes uma vez que reúnem lagos, oceano e montanhas.

Baseado nos resultados, é possível concluir que o algoritmo EMVS é mais que 3.5 vezes mais rápido que o WRF_VS além de ser capaz de processar terrenos bem grandes (maiores que 6 GB) enquanto que o WRF_VS se limita a terrenos de 2 GB (de acordo com a configuração da máquina utilizada para os testes). Porém, é importante dizer que quando o terreno é pequeno (isto é, quando ele cabe na memória interna), como é o caso do terreno de 311 MB, o WRF_VS é mais rápido que o EMVS, principalmente porque o gerenciamento e a ordenação das listas em memória externa requerem um tempo considerável na execução do algoritmo, que não é amortizado nessa situação.

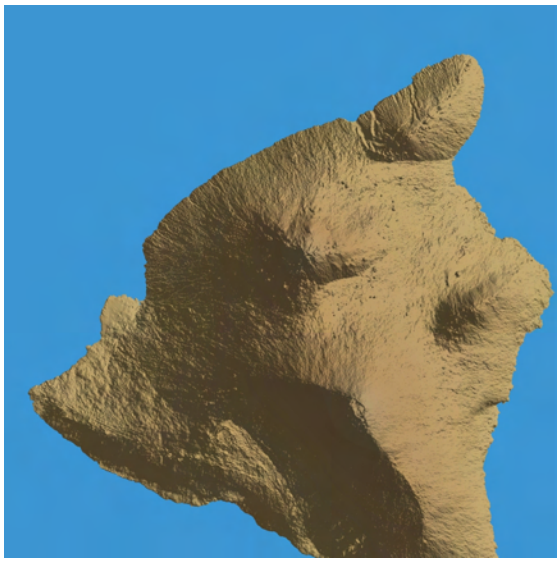


Figura 4.7: Terreno do Havaí

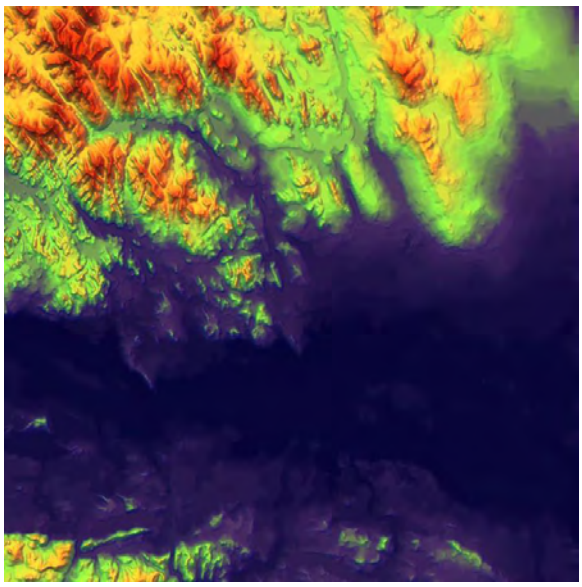


Figura 4.8: Terreno do lago *Champlain West* - Divisa EUA-Canadá

4.2.3 Comparação com Haverkort et al.

Além dos experimentos para determinar o tempo de execução do EMVS em determinados conjuntos de dados, o desempenho do EMVS foi comparado com o desempenho do algoritmo de Haverkort et al. (IO_VS). A Figura 4.10 resume essa comparação considerando os resultados disponíveis em (Haverkort et al., 2007) e assumindo como resultado do EMVS a média dos tempos de execução quando o observador está a 1

metro acima dos terrenos de 311MB, 1122MB e 4264MB, veja Tabela 4.1. Note que o EMVS é mais que 7 vezes mais rápido que o IO_VS e, vale a pena mencionar que os testes do IO_VS foram executados em um computador Power Macintosh G5 dual 2.5 GHz, 1 GB de RAM e 80 GB de HD com 7200 RPM que é significantemente mais rápido do que a máquina utilizada em nossos testes. Finalmente, como vantagem adicional, o EMVS é mais simples de implementar do que o IO_VS.

Comparando o projeto dos algoritmos EMVS e IO_VS, podemos supor que as razões principais para o melhor desempenho do EMVS são: o EMVS usa uma estrutura de dados muito simples (lista ordenada) e, após a ordenação (externa), não são realizadas atualizações (inserção e/ou remoção) na lista. Por outro lado, o algoritmo IO_VS utiliza estruturas de dados mais elaboradas (árvores binárias balanceadas) que são manipuladas por processos recursivos que envolvem operações de busca e atualização.

Tamanho do Terreno	Raio de Interesse	Tempo de execução (s) 1 GB	
		EMVS	WRF_VS
27596^2 1.45 GB	100	21	77
	500	26	81
	1000	33	97
	5000	137	438
	10000	478	1313
	15000	836	2977
32427^2 2 GB	100	26	121
	500	30	122
	1000	36	128
	5000	73	316
	10000	219	833
	15000	446	1855
55192^2 5.7 GB	100	78	-
	500	85	-
	1000	99	-
	5000	248	-
	10000	643	-
	15000	1663	-

Tabela 4.2: Comparação do tempo de execução do EMVS e do WRF_VS usando 1GB de RAM.

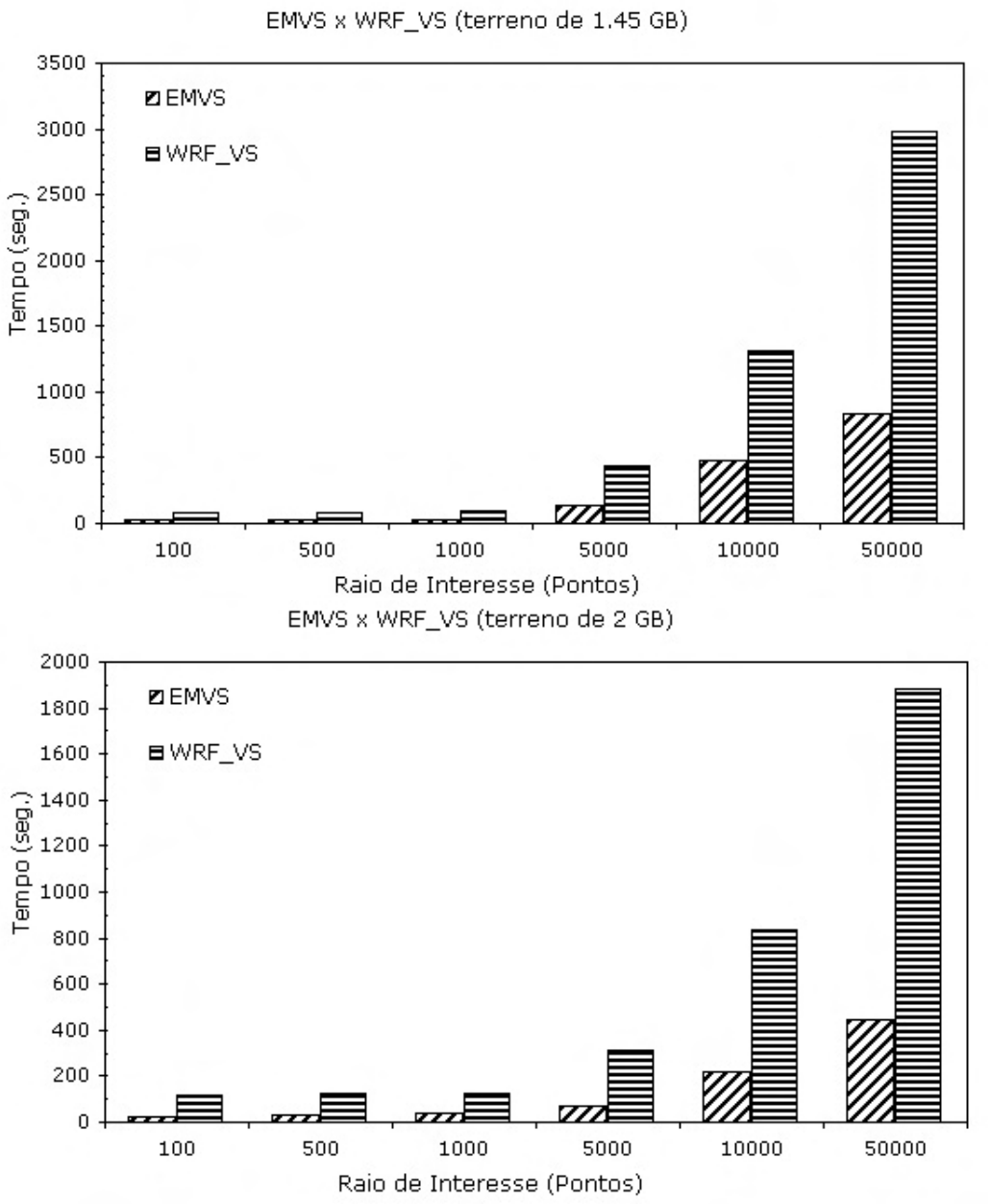
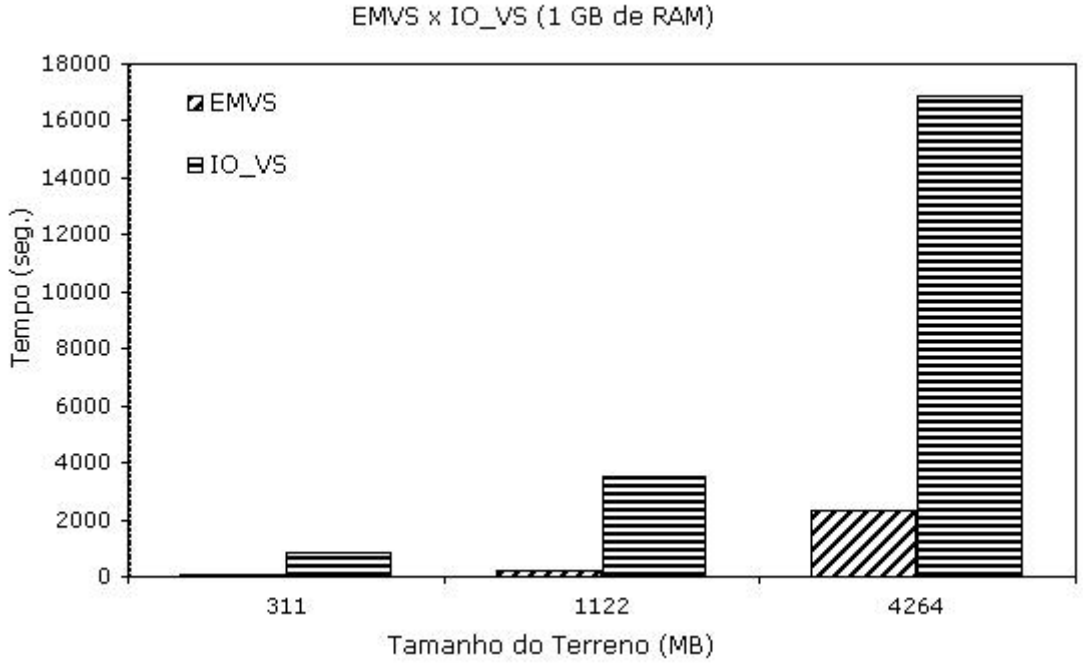
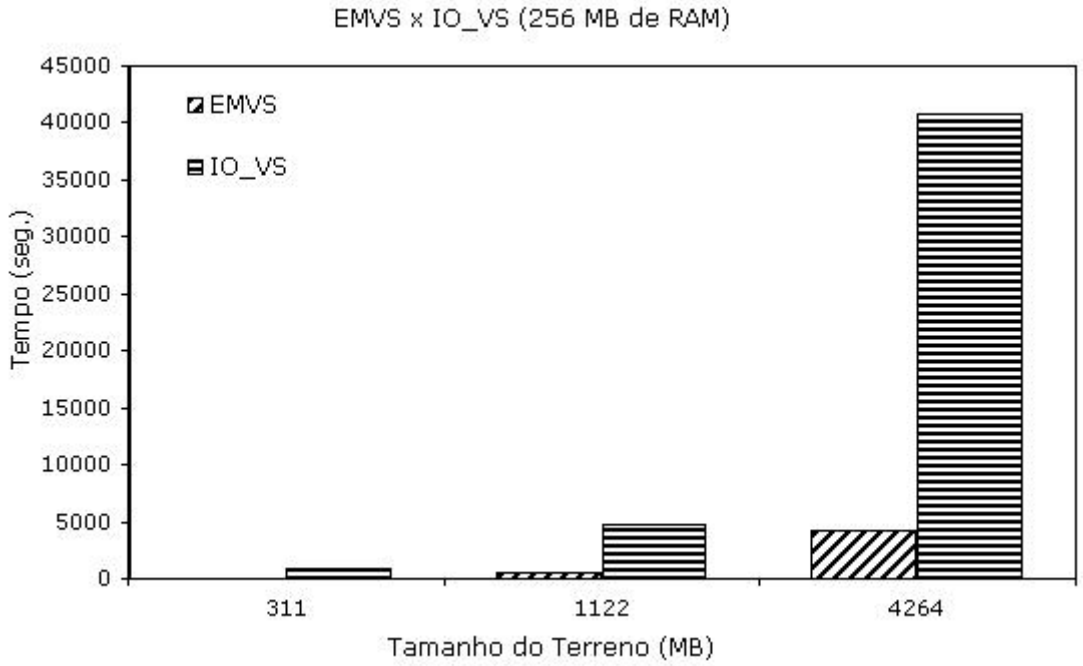


Figura 4.9: Comparação do tempo de execução do EMVS e do WRF_VS usando 1GB de RAM em terrenos de 1.45 GB e 2 GB.





Tamanho do Terreno	Tempo de execução (s)			
	1 GB de RAM		256 MB de RAM	
	EMVS	IO_VS	EMVS	IO_VS
311 MB	40	865	40	916
1122 MB	186	3546	549	4831
4264 MB	2309	16895	4133	40734

Figura 4.10: Comparação do tempo de execução do EMVS e do IO_VS usando 1 GB e 256 MB de RAM.

Conclusões Gerais

Nesta dissertação, mostramos como a evolução nas tecnologias de coleta de dados tem influenciado os algoritmos existentes. O aumento na quantidade de informações disponíveis tem superado o aumento dos recursos computacionais necessários para manipulação desse grande grande volume de dados e, portanto, novas técnicas, que aumentem a escalabilidade dos algoritmos, têm sido pesquisadas. Um aplicação bastante afetada pelo aumento de informação é o cálculo de visibilidade em terrenos, mais especificamente o cálculo do *viewshed*. Para essa aplicação, duas técnicas principais podem ser aplicadas para que dados maiores que a memória interna possam ser processados: a simplificação e os algoritmos em memória externa. A simplificação de terrenos consiste em diminuir a resolução dos terrenos retirando deles informações que, de acordo com algum critério, tentam não afetar a visibilidade dos demais pontos. Essa técnica, porém, modifica os dados originais e pode causar resultados insuficientes para a definição da visibilidade do terreno. Com relação a esse aspecto, os algoritmos em memória externa se tornam a melhor opção para o cálculo da visibilidade, já que conseguem processar todo o terreno em um tempo aceitável, pois diminuem o número de operações de entrada e saída ao manipular os dados em memória externa.

Esse problema se tornou a principal motivação para a definição do objetivo da dissertação: o desenvolvimento de um algoritmo em memória externa, o EMVS, para o cálculo de *viewshed* em terrenos. Uma outra motivação advém da importância das aplicações que utilizam o cálculo do *viewshed* envolvendo áreas como planejamento ambiental, navegação de veículos autônomos, monitoramento militar(Nagy, 1994; Li et al., 2005; Franklin e Ray, 1994).

Apresentamos, no capítulo 3, a descrição e a implementação do EMVS incluindo a descrição de como o algoritmo faz uso da *STXXL*, a biblioteca de estrutura de dados e algoritmos baseada na *STL* para manipulação de dados em memória externa. Ainda no capítulo 3 apresentamos a complexidade relacionada ao número de operações de E/S realizadas pelo algoritmo.

Todos os critérios e algoritmos utilizados para atestar a eficiência do EMVS

foram apresentados no capítulo 4. Nesse mesmo capítulo, descrevemos o algoritmo de Haverkort et al. (Toma et al., 2007) para cálculo de *viewshed* em memória externa. Nessa descrição é possível perceber que o algoritmo EMVS é bem mais simples que o algoritmo de Haverkort et al., que utiliza um processo recursivo envolvendo o paradigma *distribution sweep*. Além disso, a partir dos resultados experimentais e comparativos concluímos que o EMVS é mais que 7 vezes mais rápido que o algoritmo de Haverkort et al.. O EMVS também foi comparado ao algoritmo de Franklin e Ray (Franklin, 2002) com o objetivo de mostrar sua escalabilidade. Nesse caso, o EMVS se mostrou 3.5 vezes mais rápido, além de conseguir processar terrenos bem maiores que a memória interna, enquanto o algoritmo de Franklin e Ray se limitou a terrenos de 2GB.

A partir de todos os experimentos realizados concluímos que o EMVS é bastante simples e capaz de calcular o *viewshed* em terrenos muito maiores que a memória interna (maiores que 6GB) em um tempo menor do que os outros algoritmos existentes.

Trabalhos Futuros

Um problema importante envolvendo o cálculo do *viewshed* é o posicionamento de múltiplos observadores, que tem como objetivo encontrar o número mínimo de observadores necessários para "cobrir visualmente" uma área específica do terreno, ou então encontrar a área máxima coberta por um dado número de "observadores". Mais especificamente, dado um conjunto de observadores, o objetivo desse problema é posicionar estes observadores de modo a maximizar o índice de visibilidade de todos eles, isto é, o objetivo é maximizar a área coberta visualmente por todos os observadores (Franklin e Vogt, 2004a; Franklin e Vogt, 2006).

O índice de visibilidade de um ponto v, denotado por $vix(v)$, é dado pelo número de pontos que são visíveis a partir de v e, portanto, o cálculo de $vix(v)$ depende do cálculo do *viewshed* dos pontos do terreno. É fácil perceber que a obtenção do índice de visibilidade de vários pontos do terreno exigiria um esforço computacional considerável. Como o cálculo exato do posicionamento de múltiplos observadores necessita do índice de visibilidade de todos os pontos do terreno, essa tarefa se torna impraticável. Sharir (Cole e Sharir, 1989) demonstrou que este problema é NP-Difícil e portanto os esforços se concentram em desenvolver aproximações para o problema. Isto é, obter soluções que sejam próxima da ótima, no caso, utilizar um número de observadores próximo do número mínimo ou obter uma área de cobertura próxima do

valor máximo.

Esse problema e suas variações já foram extensivamente explorados por diversos autores (De Floriani et al., 1986; Goodchild e Lee, 1989; Lee, 1991; Kim et al., 2004; Franklin e Vogt, 2004b). Trabalhos recentes têm relacionado o problema de posicionamento de múltiplos observadores com o problema de vigilância visual que consiste em garantir a cobertura visual máxima de uma área com o número mínimo de equipamentos de vigilância (radares, torres de telecomunicação, torres de observação, sensores, etc.) (Murray et al., 2007; Rana, 2005). O problema também tem sido utilizado para posicionamento automático de antenas em redes sem fio (Ben-Shimol et al., 2007). Além disso, Franklin et al. (Tracy et al., 2007) usam este problema para planejamento de rotas em terrenos compactados.

Devido à tamanha importância desse problema, o objetivo de um trabalho futuro é propor um algoritmo que resolva o problema de posicionamento de múltiplos observadores em terrenos armazenados em memória externa. A idéia principal é projetar um algoritmo aproximado para posicionar "quase" o número mínimo de observadores necessários para cobrir "quase" todo o terreno. Mais precisamente, a idéia é desenvolver um algoritmo que determine o posicionamento de um número de observadores próximo do número mínimo de modo que uma certa porcentagem do terreno seja visível por pelo menos um dos observadores.

Apêndice A

O Algoritmo de Haverkort et al.

O algoritmo de Haverkort et al. (Haverkort et al., 2007) foi projetado para solucionar o problema de computar o *viewshed* em MDEs em memória externa. Esse algoritmo é uma adaptação do método proposto por Van Kreveld (Van Kreveld, 1996) alcançada com a utilização da varredura por distribuição.

Dado um terreno representado por uma matriz de elevação T de dimensões $n \times n$, a idéia geral na varredura por distribuição é dividir a matriz de elevação em $O(M/B)$ e $\Omega((M/B)^{\epsilon})$ subconjuntos, cada um contendo o mesmo número de células. Daí, o algoritmo decompõe a solução em duas partes: uma que pode ser encontrada através da recursão dentro de cada subconjunto e outra que envolve a interação entre subconjuntos. A parte recursiva consiste em resolver o problema recursivamente em cada subconjunto. A recursão finaliza quando o subconjunto for pequeno o bastante para caber na memória interna (o caso base) o que ocorre após $O(\log_{M/B} n)$ iterações. O desafio do algoritmo é realizar a interação entre os diversos subconjuntos, o que é realizado através da combinação de uma varredura radial e uma varredura concêntrica.

O caso base do algoritmo recursivo é uma adaptação do algoritmo de Van Kreveld para que o processamento das células possa ocorrer em memória interna com o mínimo de operações de E/S possíveis. A primeira mudança no algoritmo é realizada acrescentando aos eventos a informação sobre a elevação de cada célula para que a matriz de elevação não seja acessada após a construção da lista de eventos. Uma outra mudança é realizada através do armazenamento da visibilidade das células em uma lista ordenada pela ordem de varredura sendo que posteriormente essa lista é ordenada na ordem da matriz antes de completar a varredura. Com essas duas modificações o algoritmo não precisa carregar a matriz de elevação e a matriz com o *viewshed* calculado na memória interna. Assumindo que a lista de eventos seja armazenada em uma estrutura em memória externa, o algoritmo pode usar toda a memória para

armazenar a estrutura de células ativas.

O algoritmo realiza a varredura por distribuição da seguinte forma: inicialmente, ocorre a divisão da matriz de elevação em $O(M/B)$ subconjuntos radiais ao redor do observador. Uma célula pode ser classificada como *narrow* se ultrapassar os limites de, no máximo, um setor radial, ou *wide* se ultrapassar os limites de pelo menos dois setores e ocupar pelo menos um setor completamente, veja figuras A.1. Em cada nível de recursão o algoritmo determina como as células *wide* afetam a visibilidade das células dos setores ocupados por elas e computa a matriz de visibilidade recursivamente em cada setor. A recursão acaba quando o número de células se torna pequeno o bastante para ser executado o caso base (quando a estrutura de células ativas pode ser mantida em memória interna).

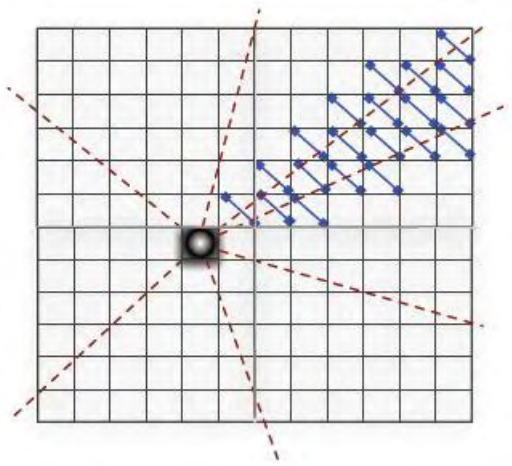


Figura A.1: Varredura por distribuição. A matriz é dividida em M/B setores radiais; células podem cruzar zero, um ou mais limites dos setores

Mais especificamente, o algoritmo varre a matriz de elevação para identificar os três eventos associados às células. Para cada célula Q, são determinados os pontos $v_1 = (r_1, \theta_1)$, o primeiro ponto de Q a ser tocado pela linha de varredura, $v_2 = (r_2, \theta_2)$, o último ponto de Q a ser tocado pela linha de varredura e o centro $q = (r_q, \theta_q)$ de Q, representados em coordenadas polares com relação ao observador p. Durante a varredura são criadas duas listas E_r e E_c. A lista E_r consiste de todos os pontos de eventos da matriz. A lista E_c contém duas cópias de cada célula Q: uma cópia marcada como *consulta* e armazenada com o ponto q correspondente ao centro da célula; e uma cópia marcada como *obstáculo* e armazenada com os pontos v_1 e v_2. Com cada célula também é armazenada a altura acima do horizonte de seu ponto central. Após a varredura de todas as células, a lista E_r é ordenada de acordo com a ordem radial ao redor do observador e as células em E_c de acordo com a distância de

seu centro até o observador.

Dadas as duas listas ordenadas E_r e E_c, o algoritmo prossegue em duas fases: uma varredura radial que processa os eventos na lista E_r, na ordem determinada pelo ângulo polar com relação ao observador, e uma varredura concêntrica, que processa os eventos da lista E_c na ordem crescente da distância entre os eventos e o observador. A varredura radial é usada para particionar os eventos em aproximadamente M/B setores de mesmo tamanho. A varredura concêntrica é usada para processar obstáculos *wide* e distribuir eventos de consulta e obstáculos *narrow*. Processando os eventos concentricamente é possível determinar se existe algum obstáculo mais próximo do observador que possa ocultar o centro da célula de consulta.

(a) (b)

Figura A.2: Segmentos correspondem a células (a) *narrow* e (b) *wide*

Na varredura radial, inicialmente são determinados os M/B setores de tal forma que cada setor contenha $O(n/(M/B))$ pontos de eventos. Isto é realizado através da ificação dos pontos de borda dos setores na lista E_r. Enquanto ocorre essa identificação, para cada setor, também é computada a lista de eventos do setor em ordem radial. Essa lista será usada posteriormente durante a recursão para determinar partições radiais dentro de cada setor.

Na varredura concêntrica, ocorre a varredura e processamento dos eventos (consultas e obstáculos) em E_c na ordem crescente da distância ao observador e, para cada setor é construída uma lista de eventos. Para realizar esse processo de forma eficiente em relação às operações de E/S, é mantido, para cada setor, um buffer de um bloco de dados na memória. Inicialmente os blocos estão todos vazios. Eles serão preenchidos com eventos durante a varredura - quando o buffer estiver cheio, seu conteúdo é levado ao disco e o buffer é esvaziado.

Além disso, para cada setor S são mantidas são mantidas as seguintes informações em memória: os dois raios que limitam o setor e uma variável que armazena a maior altura acima do horizonte dentre todos os obstáculos *wide* que cobrem com-

pletamente S e que foram atingidas pela varredura concêntrica, chamada de $High_S$. Inicialmente $High_S$ é dado por $-\infty$ para cada setor.

A varredura concêntrica prossegue da seguinte forma: o algoritmo percorre os eventos em E_c de acordo com a ordem crescente da distância ao observador. Eventos podem ser consultas ou obstáculos. Se o evento é uma célula de consulta Q, o algoritmo determina qual setor S contém seu centro e verifica se a altura de Q é no mínimo igual a $High_S$. Se verdadeiro, o algoritmo escreve Q no buffer de eventos de S. Caso contrário, o algoritmo coloca Q na lista de pontos não-visíveis e não a copia para nenhum buffer de eventos. Se o evento encontrado for um obstáculo E, ele pode interceptar diversos setores.

Para cada setor S interceptado por E, mas que não for completamente coberto por E, o algoritmo verifica se a altura de E acima do horizonte é maior que $High_S$. Se verdadeiro, E é escrito no buffer de eventos de S. Caso contrário, E é ignorado para esse setor, pois ele não pode ocultar nenhuma célula de consulta em S que não tenha sido ainda ocultada pelo obstáculo *wide* que gerou o valor de $High_S$.

Para cada setor S totalmente coberto por E (isto é, E toca ou intercepta todos os dois raios que delimitam S), o valor de $High_S$ é atualizado pelo maior valor dentre S e a altura acima do horizonte de E. Como resultado, todas as células de consulta em S que são ocultadas por E serão removidas do *stream* de eventos de S.

Após completar a varredura concêntrica, cada setor é processado recursivamente. Durante a recursão, células são marcadas como não visíveis e descartadas. Todas as células que são visíveis são mantidas até o nível final de recursão, que finaliza, como dito anteriormente, quando o algoritmo pode ser executado mantendo a estrutura ativa em memória. Conforme descrito em (Haverkort et al., 2007), o algoritmo possui complexidade de $O(sort(n^2))$ operações de E/S.

Referências Bibliográficas

Aggarwal, A. e Plaxton, G. (1994) Optimal parallel sorting in multi-level storage, In: *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pp. 659–668.

Aggarwal, A. e Vitter, J. S. (1988) The input/output complexity of sorting and related problems, Communications of the ACM, **9**:1116–1127.

Aggarwal, P.; Arge, L. e Yi, K. (2006) I/o-efficient batched union-find and its applications to terrain analysis, In: *Proceedings of ACM Symposium of Computational Geometry*.

Aho, A. V.; Hopcroft, J. E. e Ullman, J. D. (1974) The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading.

Arge, L.; Barve, R.; Toma, L. e Wickeremesinghe, R. (2002a) Tpie user manual and reference.

Arge, L.; Hinrichs, K. H.; Vahrenhold, J. e Vitter, J. S. (2002b) Efficient bulk operations on dynamic r-trees, Algorithmica, **33**(1):104–128.

Arge, L.; Procopiuc, O. e Vitter, J. S. (2002c) Implementing i/o-efficient data structures using tpie, In: *Proceedings of the European Symposium on Algorithms*, pp. 88–100.

Arge, L.; Chase, J. S.; Halpin, P.; Toma, L.; Vitter, J. S.; Urban, D. e Wickremesinghe, R. (2003) Efficient flow computation on massive grid terrains, GeoInformatica, **7**:283–313.

Arge, L.; Danner, A.; Haverkort, H. J. e Zeh, N. (2006) I/o-efficient hierarchical watershed decomposition of grid terrains models, In: *Proceedings of Symposium on Spatial Data Handling*.

Atallah, M. (1983) Dynamic computational geometry, In: *Proceedings of IEEE Symposium on Foundations of Computer Science*, pp. 92–99.

Barve, R. D. e Vitter, J. S. (March/April 2002) A simple and efficient parallel disk mergesort, ACM Transactions on Computer Systems, **35**(2):189–215.

Barve, R. D.; Grove, E. F. e Vitter, J. S. (1997) Simple randomized mergesort on parallel disks, Parallel Computing, **23**(4):601–631.

Ben-Moshe, B.; Ben-Shimol, Y. e Y. Ben-Yehezkel, A. Dvir, M. S. (2007) Automated antenna positioning algorithms for wireless fixed-access networks, Journal of Heuristics, **13**(3):243–263.

Ben-Shimol, Y.; Ben-Moshe, B.; Ben-Yehezkel, Y.; Dvir, A. e Segal, M. (2007) Automated antenna positioning algorithms for wireless fixed-access networks, Journal of Heuristics, **13**(3):243–263.

Berg, M.; Kreveld, M. V.; Overmars, M. e Schwarzkopf, O. (2000) Computational Geometry Algorithms and Applications, Springer-Verlag, Germany.

Bespamyatnikh, S.; Chen, Z.; Wang, K. e Zhu, B. (2001) On the planar two-watchtower problem, In: *Proceedings of 7th International Computing and Combinatorics Conference*, pp. 121–130.

Blelloch, G. E. (1990) Vector Models for Data-Parallel Computing, MIT Press, Cambridge.

Bresenham, J. (1965) Algorithm for computer control of a digital plotter, IBM Systems Journal, **4**(1):25–30.

Burrough, P. A. (1986) Principles of Geographical Information Systems for Land Resources Assessment, Clarendon Press, Oxford.

Caldwell, D. R.; Mineter, M. J.; Dowers, S. e Gittings, B. M. (2003) Analysis and visualization of visibility surfaces, In: *Proceedings of the 7th International Conference on GeoComputation*, University of Southampton, UK.

Cole, R. e Sharir, M. (1989) Visibility problems for polyhedral terrains, Journal of Symbolic Computation, **7**:11–30.

De Floriani, L. e Magillo, P. (1995) Horizon computation on a hierarchical terrain model, The Visual Computer: An International Journal of Computer Graphics, **1**(3):219–250.

De Floriani, L. e Magillo, P. (2003) Algorithms for visibility computation on terrains: a survey, Environment and Planning B : Planning and Design, **30**:709–728.

De Floriani, L.; Falcidieno, B.; Pienovi, C.; Allen, D. e Nagy, G. (1986) A visibility-based model for terrain features, In: *Proceedings of 2nd International Symposium on Spatial Data Handling*, pp. 235–250.

De Floriani, L.; B., F.; G., N. e C., P. (1989) Polyhedral terrain description using visibility criteria, Rel. Téc., Institute for Applied Mathematics, National Research Council, Genova - Italy.

De Floriani, L.; B., F.; G., N. e C., P. (1991) On sorting triangles in a delaunay tesselation, Algorithmica, **6**:522–532.

De Floriani, L.; Montani, C. e Scopigno, R. (1994) Parallelizing visibility computations on triangulated terrains, International Journal of Geographical Information Systems, **8**(6):515–532.

De Floriani, L.; Puppo, E. e Magillo, P. (1999) Applications of computational geometry to geographic information systems, In: J. R. Sack e J. Urrutia, eds., *Handbook of Computational Geometry*, pp. 303–311, Elsevier Science.

Dementiev, R.; Kettner, L. e Sanders, P. (2005) Stxxl : Standard template library for xxl data sets, Rel. Téc., Fakultat fur Informatik, Universitat Karlsruhe, http://stxxl.sourceforge.net/.

DeWitt, D. J.; Naughton, J. F. e Schneider, D. A. (December 1991) Parallel sorting on a shared-nothing architecture using probabilistic splitting, In: *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pp. 280–291.

Eidenbenz, S. (2002) Approximation algorithms for terrain guarding, Information Processing Letters, **82**(2):99–105.

Felgueiras, C. A. (2001) Modelagem numérica de terreno, In: G. Câmara; C. Davis e A. M. V. Monteiro, eds., *Introdução à Ciência da Geoinformação*, vol. 1, INPE, São José dos Campos.

Fisher, P. (1996) Extending the applicability of viewsheds in landscape planning, Photogrammetric Engeneering and Remote Sensing, **62**(11):1297–1302.

Franklin, W. R. (2002) Siting observers on terrain, In: Springer-Verlag, ed., *D. Richardson and P. van Oosterom editors, Advances in Spatial Data Handling: 10th International Symposium on Spatial Data Handling*, pp. 109–120.

Franklin, W. R. e Ray, C. (1994) Higher isn't necessarily better - visibility algorithms and experiments, In: *Proceedings of 6th Symposium on Spatial Data Handling*, Edinburgh, Scotland.

Franklin, W. R. e Vogt, C. (2004a) Efficient multiple observer siting on large terrain cells, In: *Proceedings of 3rd International Conference on Geographic Information Science*, University of Maryland.

Franklin, W. R. e Vogt, C. (2004b) Multiple observer siting on terrain with intervisibility or lo-res data, In: *Proceedings of XXth Congress, International Society for Photogrammetry and Remote Sensing, Istanbul*.

Franklin, W. R. e Vogt, C. (2006) Tradeoffs when multiple observer siting on large terrain cells, In: *Proceedings of 12th International Symposium on Spatial Data Handling*.

Gibson, G. A.; Vitter, J. S. e Wilkes, J. (1996) Report of the working group on storage i/o issues in large-scale computing, ACM Computing Surveys, **28**(4):779–793.

Goodchild, M. F. e Lee, J. (1989) Coverage problems and visibility regions on topographic surfaces, Annals of Operations Research, **18**:175–186.

Goodrich, M. T.; Tsay, J. J.; Vangroff, D. E. e Vitter, J. S. (1993) External-memory computational geometry, In: *Proceedings of IEEE Symposium on Foundations of Computer Science*, pp. 714–723.

Haverkort, H.; Toma, L. e Zhuang, Y. (2007) Computing visibility on terrains in external memory, In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments / Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALCO)*.

Hazel, T.; Toma, L.; Wickremesinghe, R. e Vahrenhold, J. (2006) Terracost: A versatile and scalable approach to computing least-cost-path surfaces for massive grid-based terrains, In: *Proceedings of ACM Symposium on Applied Computing*, pp. 52–57.

Hershberger, J. (1989) Finding the upper envelope of n line segments in $o(nlogn)$ time, Information Processing Letters, **33**:169–174.

Jense, J. R. (2007) Lidar remote sensing, In: J. R. Jense, ed., *Remote Sensing of the Environment: An Earth Resource Perspective*, Prentice Hall, COLCAR.

Katz, M. J.; Overmars, M. H. e Sharir, M. (1991) Efficient hidden surface removal for objects with small union size, In: *7th ACM Symposium on Computational Geometry*, pp. 31–40, New York, ACM Press.

Kim, Y. H.; Rana, S. e Wise, S. (2004) Exploring multiple viewshed analysis using terrain features and optimisation techniques, Computers and Geosciences, **30**:1019–1032.

Kirkpatrick, D. G. e Seidel, R. (1986) The ultimate planar convex hull algorithm, SIAM J. Comput, **15**:287–299.

Knuth, D. E. (1998) Art of Computer Programming, Volume 3: Sorting and Searching, Addison–Wesley, Reading, 2º edic..

Kumler, M. P. (1994) An intensive comparison of triangulated irregular network (tins) and digital elevation model (dems), Cartographica, **31**(2):monograph 45.

Lee, J. (1991) Analyses of visibility sites on topographic surfaces, International Journal of Geographical Information Systems, **5**:413–429.

Lee, J. e Stucky, D. (1998) On applying viewshed analysis for determining least-cost paths on digital elevation models, Journal of Geographical Information Science, **12**:891–905.

Li, Z.; Zhu, Q. e Gold, C. (2005) Digital Terrain Modeling - principles and methodology, CRC Press, Taylor & Francis.

Google Earth online database of satellite images (2009) Disponível em: <http://earth.google.com/>. Acesso em: 20 mar. 2009.

NASA's Earth Observing System (EOS) (2009) Nasa goddard space flight center, Disponível em : <http://eospso.gsfc.nasa.gov/>. Acesso em: 20 mar. 2009.

NASA's Shuttle Radar Topography Mission (SRTM) (2009) Nasa goddard space flight center, Disponível em : <hhttp://www2.jpl.nasa.gov/srtm/>. Acesso em: 20 mar. 2009.

TerraServer-USA: Microsoft's online database of satellite images (2009) Disponível em: <http://terraserver.microsoft.com/>. Acesso em: 20 mar. 2009.

Mills, K.; Fox, G. e Heimbach, R. (1992) Implementing an intervisibility analysis model on a parallel computing system, Computer & Geosciences, **18**(8):1047–1054.

Moore, I. D.; Turner, A. K.; Wilson, J. P.; Jenson, S. K. e E., B. L. (1993) Gis and land-surfacesubsurface process modeling, In: M. F. Goodchild; B. O. Parks e L. T. Steyaert, eds., *Environmental Modeling with GIS*, pp. 196–230, University Press, Oxford.

Murray, A. T.; Kima, K.; Davisb, J. W.; Machirajub, R. e Parentb, R. (2007) Coverage optimization to support security monitoring, Computers, Environment and Urban Systems, **31**:133–147.

Nagy, G. (1994) Terrain visibility, Computers and Graphics, **18**:763–773.

Preparata, F. P. e S., V. J. (1992) A simplified technique for hidden-line elimination in terrains, In: A. Finkel e M. Jantzen, eds., *Lecture Notes in Computer Science 577*, pp. 135–144, Springer-Verlag, Berlin.

Rallings, P. J.; Ware, J. A. e Kidner, D. B. (1998) Parallel distributed viewshed analysis, In: *Proceedings of ACM Symposium on Geographic Information Systems*, pp. 151–156.

Rana, S. (2005) Use of gis for planning visual survaillance installations, In: *Proceedings of ESRI Homeland Security Summit.*

Reif, J. e Sen, S. (1988) An efficient output-sensitive hidden surface removal algorithm and its parallelization, In: *4th ACM Symposium on Computational Geometry*, pp. 193–200, New York, ACM Press.

Schewchuk, J. R. (1996) Triangle: Engineering a 2d quality mesh generator and delaunay triangulation, In: *Proceedings of 1st Workshop on Applied Computational Geometry*, pp. 124–133, Philadelphia, Pennsylvania, Anais da ACM, 1996.

Shapira, A. (1990) Visibility and Terrain Labeling, Dissertação(mestrado), Rensselaer Polytechnic Institute, Troy, NY.

Sorensen, P. e Lanter, D. (1993) Two algorithms for determining partial visibility and reducing data structure indiced error in viewshed analysis, Photogrammetric Engeneering and Remote Sensing, **59**(7):1129–1132.

Stewart, A. J. (1998) Fast horizon computation at all points of a terrain with visibility and shading applications, In: *Proceedings of IEEE Transactions on Visualization Computer Graphics*, pp. 82 – 93.

Teng, Y. A.; Mount, D.; Puppo, E. e Davis, L. S. (1995) Parallelizing an algorithm for visibility on polyhedral terrain, International Journal of Geographical Information Systems, **7**(1–2):75–84.

Toma, L.; Arge, L. e Vitter, J. S. (2001) I/O-efficient algorithms for problems on grid-based terrains, ACM J. Experimental Algorithmics, **6**:1.

Toma, L.; Haverkort, H. e Zhuang, Y. (2007) Computing visibility on terrains on external memory, In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments.*

Tracy, D. M.; Franklin, W. R.; Cutler, B.; Andrade, M.; Inanc, M. e Xie, Z. (2007) Smugglers and border guards - the geostar project at rpi, In: *Proceedings of 15th ACM International Symposium onAdvances in Geographic Information Systems.*

Van Kreveld, M. (1996) Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals, In: *Proceedings of Symposium on Spatial Data Handling*, pp. 15–27.

Van Leeuwen, J. (1990) Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, Elsevier, COLCAR.

Vengroff, D. E. e Vitter, J. S. (1995) I/o-effcient scientific computation using tpie, In: *Proceedings of IEEE Symposium on Parallel and Distributed Computing.*

Verdin, K. L. e Verdin, J. P. (1999) A topological system for delineation and codification of the earth's river basins, Journal of Hydrology, **218**:1–12.

Vitter, J. S. (2001) External memory algorithms and data structures: Dealing with massive data, ACM Computing Surveys, **33**(2):209–271.

Vitter, J. S. e Shriver, E. A. M. (1994) Algorithms for parallel memory i: Two-level memories, Algorithmica, **12**(2-3):110–147.