SALLES VIANA GOMES DE MAGALHÃES

MÉTODOS PARA O POSICIONAMENTO DE RECURSOS EM TERRENOS REPRESENTADOS POR GRADES REGULARES

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
2010

SALLES VIANA GOMES DE MAGALHÃES

MÉTODOS PARA O POSICIONAMENTO DE RECURSOS EM TERRENOS REPRESENTADOS POR GRADES REGULARES

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: 15 de Dezembro de 2010.

José Elias Claudio Arroyo (Co-Orientador)	Vladimir Oliveira Di Iorio (Co-Orientador)
Carlos Antonio Álvares Soares Ribeiro	Ricardo dos Santos Ferreira

Marcus Vinicius Alvim Andrade
(Orientador)

AGRADECIMENTOS

Agradeço primeiro a Deus por me guiar em todos os momentos da minha vida e me ajudar a vencer cada novo desafio.

Aos meus pais, Eliezer e Ana, por estarem sempre ao meu lado me incentivando, sobretudo durante o tempo em que estive em Viçosa. Ao meu avô Alcy, por ter incentivado a minha curiosidade pela ciência. Agradeço também a toda minha família, que sempre trouxe alegria e motivação para a minha vida.

Aos amigos de Viçosa que sempre me propiciaram momentos de descontração durante o período em que estudei na Universidade Federal de Viçosa.

Gostaria de agradecer também a todos os professores e funcionários do Departamento de Informática, que não só contribuíram para minha formação profissional como também sempre estiveram dispostos a conversar e me ajudar durante o período em que estive na universidade.

Em especial, faço um agradecimento ao meu orientador Prof. Marcus Vinicius, que me acompanhou desde a graduação até a pós-graduação dedicando seu tempo e atenção para me incentivar e auxiliar a realização das pesquisas e o desenvolvimento dessa dissertação.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

MAGALHÃES, Salles Viana Gomes de, M.Sc., Universidade Federal de Viçosa, Dezembro de 2010. **Métodos para o posicionamento de recursos em terrenos representados por grades regulares**. Orientador: Marcus Vinicius Alvim Andrade. Co-Orientadores: José Elias Cláudio Arroyo e Vladimir Oliveira Di Iorio.

O aumento da disponibilidade de dados geográficos de boa qualidade tem feito crescer a necessidade do desenvolvimento de técnicas mais avançadas para o processamento eficiente desses dados em computadores utilizando SIGs (Sistemas de Informações Geográficas). Um problema importante na área de SIG é o posicionamento de observadores em terrenos. Esse problema consiste em minimizar o número de observadores necessários para cobrir visualmente certa porcentagem do terreno. Nesse trabalho são apresentadas diferentes abordagens para a solução do problema de posicionamento de observadores. São propostas heurísticas que consideram tanto instâncias do problema que podem ser processadas na memória principal dos computadores quanto instâncias que precisam ser processadas utilizando a memória secundária. Também é proposta uma abordagem que utiliza o processamento paralelo em placas gráficas para diminuir o tempo de execução dessas heurísticas. Os testes realizados indicam que as heurísticas propostas geram soluções que utilizam até 17% menos observadores do que outro método descrito em literatura. Além disso, a abordagem que utiliza placas gráficas foi até 4 vezes mais rápida do que a abordagem que realiza processamento sequencial.

ABSTRACT

MAGALHÃES, Salles Viana Gomes de, M.Sc., Universidade Federal de Viçosa, December of 2010. **Methods for siting observers in terrains represented by regular grids**. Adviser: Marcus Vinicius Alvim Andrade. Co-Advisers: José Elias Cláudio Arroyo and Vladimir Oliveira Di Iorio.

The enormous volumn of high quality geographic data has required the development of more advanced techniques to process them efficiently in computers using GISs (Geographic Information Systems). An important problem in GIS applications is the siting of observers on terrains. This problem consists in minimizing the number of observers needed to achieve a given visual coverage of the terrain. This work presents different approaches to solve the observer siting problem. Heuristics to solve this problem in the main memory and in the external memory are presented. Also, it is proposed a parallel approach to solve this problem in graphic cards in order to decrease the heuristics processing time. Tests have shown that the proposed heuristics generates solutions uaing about 17% less observers than the solution obtained by other method described in literature. Also, the approach using graphical cards executes almost four times more quickly than the sequential approach.

Capítulo 1

Introdução

1.1 Motivação

O aumento da disponibilidade de dados geográficos de boa qualidade tem feito crescer a necessidade do desenvolvimento de técnicas mais avançadas para o processamento desses dados em computadores utilizando os SIGs (Sistemas de Informações Geográficas) que são sistemas que manipulam enorme massas de dados [22]. Tais dados podem ser obtidos de diversas formas, dentre as quais pode-se destacar: digitalização de imagens de satélites e de mapas ou medição realizada manualmente utilizando equipamentos de GPS [12; 23].

Normalmente esses dados são representados sob a forma de modelos digitais de elevação (MDEs), que são representações aproximadas do terreno real. Essas representações são montadas utilizando uma amostra de pontos do terreno nos quais são conhecidos os valores da elevação. A elevação dos pontos que não pertencem à amostra pode ser obtida utilizando um processo de aproximação baseado nos pontos conhecidos [12].

Dentre as várias aplicações na área de SIGs envolvendo terrenos, há aquelas relacionadas a questões de visibilidade. Tais aplicações utilizam o conceito de observador e alvo: o observador tem o objetivo de visualizar (observar) outros objetos (os alvos) em um terreno, sendo que os observadores possuem um limite máximo para o alcance de sua visão chamado de raio de interesse. Por exemplo, uma torre de celular pode ser considerada um observador cujo raio de interesse corresponde ao alcance do sinal da torre e cujos alvos são os usuários do serviço de telefonia. Veja a Figura 1.1: note que a linha tracejada liga a torre a um dispositivo visível a partir dela e a linha pontilhada liga a torre a um dispositivo não visível.

Com base nesse conceito, pode-se calcular o mapa de visibilidade (*viewshed*) de um ponto p do terreno que indica a região do terreno que é visível por um observador posicionado em p. Outra característica importante que também pode ser determinada a partir do processamento do terreno é o índice de visibilidade, que indica a "quantidade"

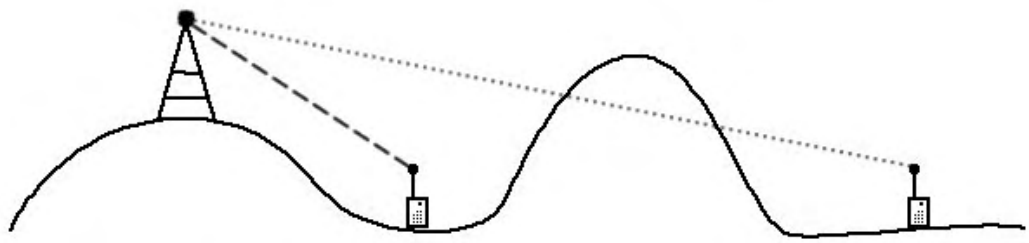


Figura 1.1. Torre de telefonia celular

de pontos que podem ser visualizados a partir de determinadas posições do terreno.

Essas aplicações possuem grande utilidade prática em várias áreas como planejamento ambiental [23], navegação de veículos autônomos [14], monitoramento militar [15].

Uma outra classe importante de problemas relacionados a terrenos são os problemas de posicionamento de facilidades (ou recursos). Tais problemas visam posicionar recursos em um terreno de modo a otimizar (maximizar ou minimizar) uma função objetivo satisfazendo certas restrições. Exemplos de restrições são o atendimento da demanda de clientes por produtos produzidos nas facilidades, o atendimento de um número mínimo de clientes, etc. Exemplos de facilidades incluem fábricas, depósitos, torres de telefonia celular, etc [4].

Em particular, pode-se considerar o problema de posicionamento de observadores em um terreno. Tais observadores podem ser torres de telefonia celular e transmissores de sinais de rádio, postos de observação, etc [16; 14]. O objetivo desse posicionamento é minimizar o número de observadores necessários para cobrir um certo percentual da área do terreno em questão, ou seja, para que certo percentual da área do terreno seja visível a partir dos observadores posicionados. Uma variação desse problema de posicionamento de observadores é o problema de maximizar a área coberta a partir

de um número fixo de observadores. Esses tipos de problemas envolvem conceitos de visibilidade em terrenos e também de otimização para o posicionamento de recursos.

Um exemplo de tal problema consiste em, dado um terreno T que representa uma cidade e um conjunto P de pontos dessa cidade onde é possível instalar torres de telefonia celular, determinar o menor número necessário de torres para se obter cobertura de sinal de celular em pelo menos 80% da cidade.

Observe que, para se realizar esse tipo de posicionamento, é necessário minimizar a redundância (áreas que são visíveis por mais de um observador) e, ao mesmo tempo, posicionar os observadores em pontos cujos índices de visibilidade sejam altos.

Note que em certos casos, como por exemplo, para o posicionamento de torres de telefonia celular, mesmo uma pequena redução na quantidade de recursos utilizados poderia representar grande economia.

1.2 Objetivos

A proposta deste trabalho é propor e implementar métodos para realizar o posicionamento de observadores em terrenos representados por modelos digitais de elevação representados por grades regulares. Os métodos propostos irão considerar 3 tipos de processamento de dados: processamento em *CPU*, processamento em memória externa e processamento em *GPUs* (*Graphics Processing Units* - Unidades de Processamento Gráfico) utilizando a tecnologia *CUDA* [28]. O processamento em memória externa tem o objetivo de posicionar observadores em terrenos que não possam ser processados na memória principal dos computadores (terrenos muito grandes). Já o processamento em *GPUs* visa aproveitar o poder de processamento paralelo delas para realizar o posicionamento de observadores (em terrenos que possam ser processados na memória da *GPU*) de forma mais rápida. Mais especificamente, pretende-se:

1. Desenvolver métodos para o posicionamento de observadores em memória interna utilizando heurísticas de otimização combinatória como a heurística *GRASP* [36].

2. Adaptar os métodos desenvolvidos para que eles utilizem processamento em *GPUs*.

3. Desenvolver métodos para a solução do problema considerando terrenos de alta resolução que não podem ser processados em memória interna.

4. Realizar testes com o objetivo de ajustar o método proposto para melhorar o desempenho e a qualidade dos resultados.

5. Realizar testes para validar os métodos.

1.3 Organização da dissertação

No Capítulo 2, são descritos os principais conceitos utilizados nesse trabalho. A Seção 2.1 apresenta as principais formas utilizadas para representar modelos digitais de terrenos. Na Seção 2.2 são descritos conceitos relacionados ao problema de posicionamento de observadores e a visibilidade em terrenos e, na Seção 2.3, são descritos os métodos propostos na literatura para resolver esses problemas. As técnicas utilizadas na literatura para solução do problema de posicionamento de recursos em terrenos são apresentadas na Seção 2.4. A Seção 2.5 cita algumas heurísticas utilizadas para resolver problemas gerais de otimização combinatória. Por fim, as seções 2.6 e 2.7 apresentam, respectivamente, conceitos relacionados a processamento de dados em memória externa e em *GPUs*.

O Capítulo 3 descreve os métodos propostos neste trabalho. A Seção 3.1 descreve detalhes das principais estruturas de dados que foram utilizadas para representar os dados de visibilidade e, na Seção 3.2, são descritas as principais operações que as heurísticas propostas aplicam a essas estruturas de dados para realizar o posicionamento de observadores. Na Seção 3.4, são apresentadas as heurísticas propostas para o problema de posicionamento de observadores. A Seção 3.5 apresenta detalhes da implementação em *GPU* das heurísticas descritas na Seção 3.4. Finalmente, a Seção 3.6 descreve o método de posicionamento de observadores em memória externa proposto nesse trabalho.

O Capítulo 4 descreve os experimentos realizados para avaliar a qualidade das soluções e o tempo de processamento dos métodos propostos.

Finalmente, no Capítulo 5, são apresentadas as conclusões obtidas nesta dissertação e sugestões de trabalhos futuros.

Vale mencionar que os resultados obtidos neste trabalho foram publicados e apresentados em importantes congressos da área. Mais especificamente, as heurísticas de posicionamento de observadores em memória interna foram publicados no artigo [25], que foi apresentado no "10^o Simpósio Brasileiro de GeoInformática" (GEOINFO 2010). A heurística para posicionamento de observadores em terrenos armazenados em memória externa foi publicada nos artigos [24] e [26] que foram apresentados, respectivamente, na "XXXV Conferência Latino Americana de Informática" (CLEI 2009) e na "10^{th} International Conference on Hybrid Intelligent Systems" (HIS 2010).

Capítulo 2

Referencial teórico

2.1 Representação digital de terrenos

Segundo Felgueiras [12], a representação no computador da superfície de um terreno é feita utilizando-se um Modelo Numérico de Terreno (*MNT*) que representa matematicamente a distribuição de um fenômeno espacial que ocorre dentro da área da superfície terrestre a ser trabalhada. Exemplos de fenômenos representados por *MNTs* incluem dados de relevo, informações geológicas, informações meteorológicas, entre outros. No caso particular desse trabalho, um *MNT* será utilizado para representar os dados de elevação de um terreno. Nesse caso, o *MNT* recebe a denominação de Modelo Digital de Elevação (*MDE*).

Um *MNT* é gerado em duas etapas: (1) amostragem dos dados e (2) geração do modelo. Na primeira etapa, um conjunto de amostras que representam a variação do fenômeno espacial de interesse é obtido (através de levantamento em campo, digitalização de imagens, levantamento a partir de dados de satélites, etc). Tal amostragem deve ser feita com o objetivo de representar bem a variação do fenômeno a ser estudado. Por exemplo, uma superamostragem de altimetria em uma região plana significa redundância de dados enquanto que uma pequena quantidade de pontos em uma região montanhosa significa escassez de informações [12].

As principais formas de amostragem de pontos são a regular e a irregular. A amostragem regular é caracterizada pela distribuição regular das coordenadas geográficas (x, y) dos pontos amostrados, ou seja, o espaçamento vertical e horizontal entre os pontos amostrados é constante. Já na amostragem irregular não existe regularidade na distribuição das amostras.

Após a amostragem dos dados, eles são utilizados para gerar a estrutura de dados do *MNT*. No caso de *MDEs*, estes normalmente são representados utilizando dois tipos de estruturas de dados: a *TIN* (*Triangulated Irregular Network* ou Malha Triangular Irregular) e a grade regular. A *TIN* representa o terreno utilizando uma malha triangular cujos vértices representam os pontos amostrados de forma irregular no terreno.

A partir desses pontos, a elevação de cada ponto p não pertencente à amostra é obtida através da interpolação bi-linear dos três vértices do triângulo cuja projeção no plano xy contém a projeção de p. Já a grade regular representa o terreno com uma matriz que contém a elevação de pontos amostrados de forma regular. Veja as Figuras 2.1 e 2.2.

Devido à sua simplicidade e à grande disponibilidade de dados nesse formato, esse trabalho irá utilizar *MDEs* no formato de grade regular. É importante observar que uma forma de representação de dados pode ser convertida na outra de maneira bastante eficiente utilizando processos de interpolação e, portanto, a escolha de um modelo de representação de dados para ser utilizado por uma aplicação *SIG* não representa uma restrição.

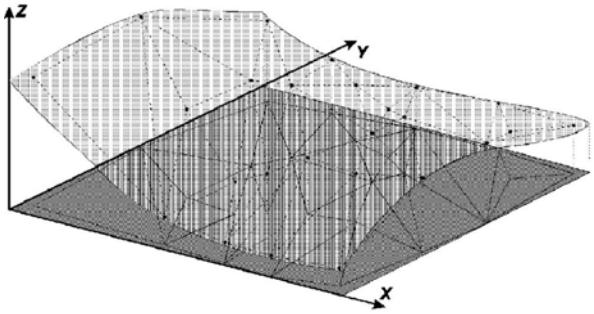


Figura 2.1. Exemplo de Malha Triangular Irregular (Fonte: Felgueiras [11])

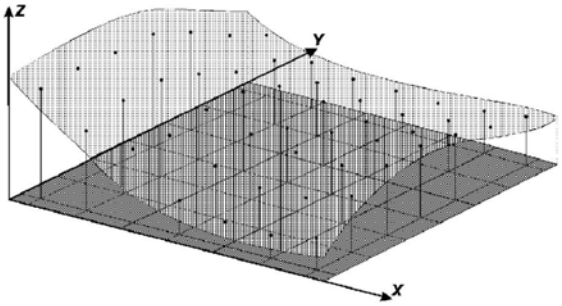


Figura 2.2. Exemplo de grade regular (Fonte: Felgueiras [11])

2.2 Conceitos sobre visibilidade

Um *observador* é um ponto no espaço a partir do qual se deseja ver ou comunicar com outros pontos, chamados de *alvos*. As notações usuais para observadores e alvos são, respectivamente, O e T. Os *pontos base* de O e T, denominados como O_b e T_b, são os

pontos do terreno que estão diretamente abaixo de O e T, respectivamente. O e T são posicionados a uma altura h acima de O_b e T_b.

O *raio de interesse*, R, de O representa o alcance da visibilidade de O. Por exemplo, se O é um rádio transmissor, R dependerá da potência do transmissor e da sensibilidade do receptor de rádio. Por conveniência, R é medido entre O_b e T_b em vez de entre O e T, o que é equivalente quando h é muito menor do que o raio da Terra.

T é *visível* a partir de O se, e somente se, $|T_b - O_b| \leq R$ e não há nenhum ponto entre O e T bloqueando o segmento de reta, chamado de Linha de Visão (ou *Line of Sight - LOS*), que liga O a T. Veja a Figura 2.3. Nessa Figura, T_1 e T_3 são visíveis a partir de O mas T_2 não é visível.

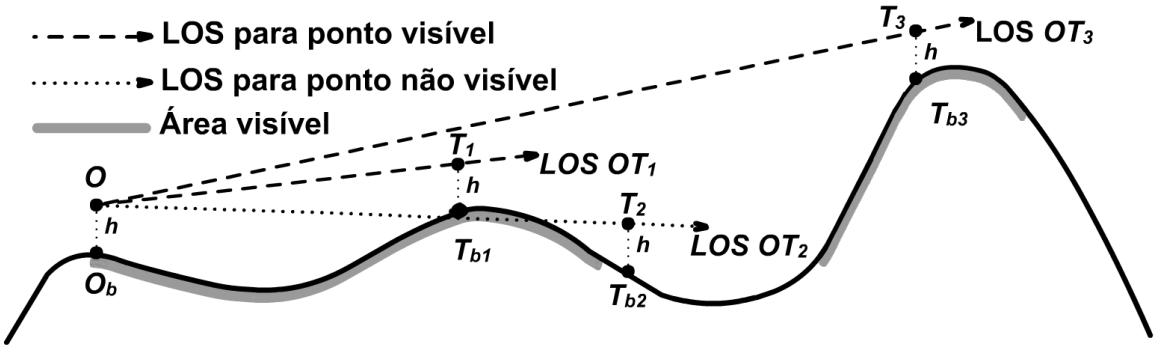


Figura 2.3. Cálculo de visibilidade utilizando uma *LOS* em um corte vertical de terreno.

O *viewshed* (ou mapa de visibilidade) de O, denotado por *VS(O)*, é o conjunto de pontos base cujos alvos correspondentes são visíveis a partir de O. Por motivos de eficiência, o *viewshed* pode ser armazenado sob a forma de uma matriz de bits codificados em palavras de máquina.

O *índice de visibilidade* de um observador O é o número de alvos que são visíveis a partir de O. Pontos com alto índice de visibilidade, em geral, são bons candidatos onde se posicionar observadores com o objetivo de maximizar a área do terreno que é vista por pelo menos um observador [15]. O índice de visibilidade pode ser obtido contando-se o número de pontos visíveis no *viewshed* do observador correspondente, mas, por questões de eficiência, normalmente é estimado utilizando-se uma amostragem de alvos no terreno.

O *viewshed acumulado* de um conjunto de observadores $\mathcal{O} = \{O_i\}$, denotado por *VS($\mathcal{O}$)*, é a união dos *viewsheds* individuais de cada observador O_i.

O *índice de visibilidade* (ou *vix - visibility index*) de um conjunto $\mathcal{O}$ de observadores, denotado por $VIX(\mathcal{O})$, é o número de alvos no terreno que são visíveis por pelo menos um observador em $\mathcal{O}$. $VIX(\mathcal{O})$ pode ser normalizado para corresponder

a uma porcentagem do terreno. O índice de visibilidade é informalmente associado à área visível ou área coberta.

Dado um conjunto de observadores $\mathcal{O}$, $VIX(\mathcal{O})$ pode ser calculado contando-se o número de pontos visíveis em *VS($\mathcal{O}$)*. Note que essa operação é bastante dispendiosa computacionalmente visto que é necessário realizar a união de *viewsheds* para gerar *VS($\mathcal{O}$)* e, então, contar o número de pontos visíveis em *VS($\mathcal{O}$)*.

O problema de *posicionamento de múltiplos observadores* consiste em otimizar a localização de um conjunto de observadores com o objetivo de maximizar o índice de visibilidade desses observadores. Esse problema possui muitas aplicações práticas como o posicionamento de torres de telefonia celular, de torres de observação de incêndios, de sistemas de radares, etc.

O problema de posicionamento de múltiplos observadores é NP-Completo [32], e pode ser reduzido ao problema clássico de cobertura de conjuntos [8] que consiste em, dado um conjunto $K = \{k_i\}$ de conjuntos, obter o menor número de conjuntos k_i tais que a união desses conjuntos seja igual à união de todos k_i. Mais precisamente, obter $C = \{c_i\} \subset K$ tal que $\cup c_i = \cup k_i$ e $|C|$ é minimizado [5].

Esse trabalho irá considerar a seguinte variação do problema de posicionamento de múltiplos observadores: minimizar o número de observadores necessários para se cobrir uma porcentagem do terreno. Mais especificamente, dado um conjunto $P = \{P_i\}$ de pontos candidados a receberem observadores, o problema consiste em encontrar o menor subconjunto $S = \{s_i\}$ de P cujo índice de visibilidade seja maior ou igual a um valor mínimo VIX_{min} definido pelo usuário. Dessa forma, a solução para o problema será representada por um subconjunto de P. Vale dizer que esse problema também é NP-Completo.

2.3 Métodos para solução de problemas de visibilidade

Como afirmado na Seção 2.2, os problemas de visibilidade formam um importante grupo de problemas envolvendo o processamento de modelos digitais de terrenos. Nesta seção serão descritas técnicas para determinação de visibilidade, cálculo de índice de visibilidade e de mapas de visibilidade.

Normalmente, o cálculo da visibilidade se baseia na seguinte estratégia: dado um terreno representado por uma matriz de elevação M e dados dois pontos O_b e T_b pertencentes a M, um alvo T e um observador O (com raio de interesse R) sendo que tanto O e T podem estar posicionados a uma dada altura acima dos pontos no terreno O_b e T_b, respectivamente, então o cálculo da visibilidade de T consiste em traçar a reta imaginária denominada linha de visão ou LOS (*line of sight*) entre os pontos O e T

e se deslocar sobre essa linha, determinando se existe algum ponto nesse caminho tal que a elevação do terreno nesse ponto seja maior do que a elevação da linha. Se não existir nenhum ponto mais alto, T é considerado visível, caso contrário T é considerado não visível. Veja a Figura 2.3, onde há um observador O posicionado em O_b e linhas de visão ligando O a alvos visíveis visíveis (posicionados em T_{b1} e T_{b3}) e a um alvo não visível (posicionado em T_{b2}).

A partir do método de determinação de visibilidade, é possível extrair o índice de visibilidade de um ponto P aplicando esse método a todos os pontos no interior do círculo de alcance visual de O (círculo de raio R centrado em P) e, então, dividindo-se o número de pontos que são considerados visíveis pelo número total de pontos do interior do círculo.

Por motivos de eficiência, geralmente o índice de visibilidade é calculado de forma aproximada. Esse cálculo consiste em amostrar uma determinada quantidade de pontos (escolhidos aleatoriamente) do círculo de alcance visual e, então, calcular a visibilidade apenas nesses pontos (nesse caso, o índice de visibilidade corresponde ao número de pontos visíveis dividido pelo número de pontos amostrados). Esse tipo de cálculo aproximado é utilizado no método de posicionamento de observadores desenvolvido por Franklin e Ray [14].

Outra característica importante de um terreno que pode ser obtida a partir da determinação de visibilidade é o *viewshed*. Uma forma direta de se obter o *viewshed* de um observador O consiste em aplicar um método de determinação de visibilidade a todos os pontos que estão no círculo de alcance visual de O. Assim, o *viewshed* de O consistirá em todos os pontos que foram considerados visíveis pelo método. Vários pesquisadores desenvolveram métodos de cálculo de *viewshed* alternativos a essa solução direta: em [35; 42] são propostos métodos mais eficientes para cálculo de viewshed em memória interna e, em [19; 3], são propostos métodos para o cálculo de *viewshed* em memória externa.

2.4 Posicionamento de recursos em terrenos

Como descrito na Seção 2.2, dado um terreno M e um conjunto P de pontos de M candidatos a receberem recursos, o problema de posicionamento de recursos consiste em determinar o menor subconjunto P' de P tal que certas restrições sejam satisfeitas.

De modo geral, o ideal é considerar que todos os pontos são candidatos a receberem recursos. No entanto, como um modelo digital de terreno pode conter uma grande quantidade de pontos, essa solução seria muito ineficiente e, portanto, é importante utilizar um método para reduzir o número de pontos candidatos. Em [14], o índice de

visibilidade de todos os pontos do terreno é utilizado para determinar os pontos que pertencerão a P (escolhem-se os pontos com maiores índices de visibilidade). Nesse caso, o método é utilizado para se resolver de forma aproximada o problema de posicionamento onde todos os pontos do terreno são aptos a receberem observadores. Em [6], um conjunto de pontos candidatos definidos pelo usuário é utilizado para resolver o problema de posicionamento de torres transmissoras de ondas de rádio (que nesse caso são os observadores).

Após a escolha dos pontos de P, é preciso determinar o menor subconjunto P' de P, ou seja, a solução com menor quantidade de candidatos que satisfaz as restrições do problema de posicionamento de recursos. No caso do problema de posicionamento de observadores, deve-se escolher P' de forma que os observadores posicionados nos pontos desse subconjunto cubram pelo menos uma determinada porcentagem do terreno (esse valor é definido pelo usuário).

Visto que esse problema é NP-Completo [32], é importante o desenvolvimento de heurísticas para a solução aproximada de tal problema. Franklin e Ray [15; 13] adaptaram uma estratégia gulosa utilizada na solução do problema de cobertura de conjuntos para gerar uma solução aproximada do problema. Mais precisamente, este método (denominado *Site*) consiste nas seguintes etapas:

1. Cálculo do índice de visibilidade: Calcula-se o índice de visibilidade aproximado de todos os pontos do terreno onde se deseja posicionar observadores.

2. Escolha dos pontos candidatos: Os k (valor definido pelo usuário) pontos com maiores índices de visibilidade são escolhidos para serem candidatos a receber observadores (esses pontos formam o conjunto P).

3. Cálculo do *viewshed*: Calcula-se o *viewshed* de todos os pontos de P.

4. Posicionamento dos observadores: Utiliza-se uma iteração gulosa na qual, a cada passo, adiciona-se à solução corrente o observador cujo mapa de visibilidade mais contribuir para o aumento da área visível do terreno. A iteração é encerrada quando se atingir a área de cobertura desejada.

O fluxograma da Figura 2.4 resume as etapas do método *Site* [15; 13].

Além da heurística gulosa, outras heurísticas de otimização combinatória podem ser utilizadas na solução desse problema. Em [4], são propostas heurísticas, como algoritmos genéticos, para a solução do problema de posicionamento de facilidades. Kim et al. [21] propõem o uso de métodos de têmpera simulada e um algoritmo genético para resolver o problema de posicionamento de observadores. Em [10], são

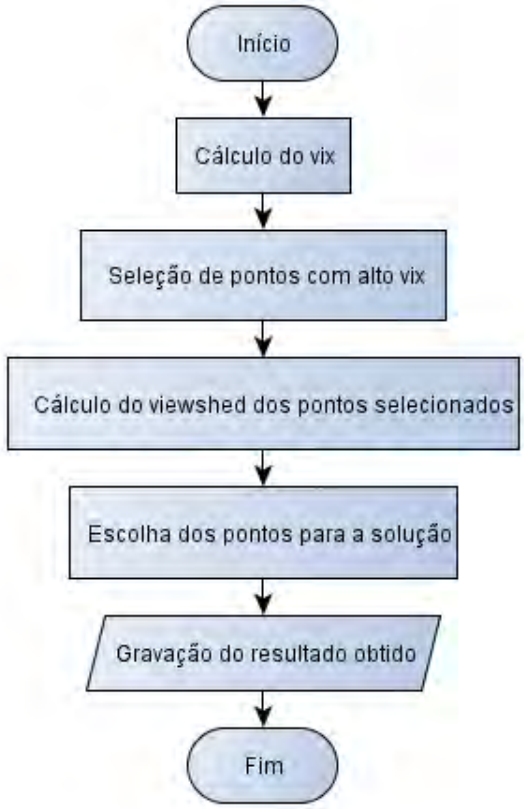


Figura 2.4. Fluxograma com a sequencia de passos utilizados pelo método *Site* [15; 13].

avaliadas heurísticas gulosas e um algoritmo genético para a solução do problema de posicionamento de torres de internet sem fio.

2.5 Uso de heurísticas para a solução de problemas de otimização combinatória

Os problemas de otimização cujas soluções exatas são difíceis de serem obtidas, geralmente são resolvidos de forma aproximada utilizando heurísticas de otimização combinatória. Exemplos de heurísticas são buscas locais e *GRASP* (*Greedy Randomized Adaptative Search Procedure*) [36], algoritmos genéticos [31] e têmpera simulada [9].

Essas heurísticas normalmente utilizam várias tentativas (iterações) para se obter uma solução que maximize (ou minimize) o valor de uma função objetivo satisfazendo certas restrições. No caso do problema de posicionamento de observadores, um exemplo de função objetivo consiste em analisar o número de observadores utilizados (nesse caso desejamos minimizar essa função) sendo que a restrição é que a área coberta pelos

observadores deve ser maior ou igual a um valor mínimo de área.

A heurística *GRASP* [36], por exemplo, utiliza uma iteração que, a cada passo, gera uma solução gulosa randomizada (uma solução que apresenta características tanto de uma solução gulosa quanto de uma solução aleatória) e, então, tenta otimizar a função objetivo dessa solução utilizando um método de busca local. Após atingir um critério de parada (por exemplo, um número máximo de iterações), o método pára e retorna a melhor solução obtida.

Outra característica importante de tais heurísticas é que elas podem ser combinadas com o objetivo de encontrar soluções melhores. Por exemplo, Resende [37] propôs uma combinação de heurísticas para resolver o problema das p-medianas (um tipo especial de problema de posicionamento de facilidades). A idéia consiste em utilizar uma heurística de busca local para melhorar as soluções obtidas pelo algoritmo genético.

Além disso, muitas heurísticas podem ser implementadas de forma paralela para obter boas soluções utilizando menos tempo de processamento. Solar et al. [40], por exemplo, proporam um algoritmo genético paralelo para a solução do problema de cobertura de conjuntos. Já Aiex et al. [2] apresentam um método *GRASP* paralelo para o problema de programação de tarefas.

2.6 Processamento de dados em memória externa

As operações em memória externa são aproximadamente 10^6 vezes mais lentas do que as que ocorrem na memória interna [11]. Isto se deve à natureza mecânica dos discos rígidos. Ao se acessar um dado no disco, há a necessidade do movimento da cabeça de gravação de modo a posicioná-la na trilha adequada. O tempo gasto para o processo descrito acima é chamado de latência. Uma vez que a cabeça de gravação está posicionada, os dados que estão gravados de modo sequencial em relação a essa posição podem ser lidos ou gravados no disco. Para melhorar o desempenho do disco, os dados são gravados em blocos de tamanho B de modo que a transferência de um bloco seja feita na mesma velocidade do que a transferência de apenas uma palavra de memória. Assim, no acesso sequencial, os dados ficam divididos em blocos consecutivos, diminuindo então o número de operações de E/S necessárias.

Para definir melhor a complexidade dos algoritmos que processam dados em memória externa, foi desenvolvido um modelo computacional denominado *PDM*, *"Parallel Disks Model"* [1] que é composto por uma memória principal (onde o processamento é realizado) de tamanho limitado M e uma memória externa de tamanho infinito formada por D discos independentes. Uma operação de E/S transfere D blocos de tamanho B

(contendo dados consecutivos) entre o disco e a memória de modo a diminuir o problema de latência[1] .

No processamento em memória externa, a complexidade do algoritmo deve ser avaliada levando-se em conta o acesso e a transferência de dados entre a memória secundária e a memória principal, visto que essas operações afetam de forma decisiva o tempo de execução do algoritmo. Portanto, a eficiência do algoritmo é medida em termos do número de operações de E/S e não em termos do uso da *CPU*. No *PDM* são definidas as seguintes ordens de complexidade:

- Ler/gravar N itens consecutivos: $\Theta(scan(N)) = \Theta\left(\frac{N}{B}\right)$
- Ordenar N itens consecutivos: $\Theta(sort(N)) = \Theta\left(\frac{N}{B}\log_{\left(\frac{M}{B}\right)}\left(\frac{N}{B}\right)\right)$

Essas complexidades podem ser "ordenadas" da seguinte forma: $\Theta(scan(N)) < \Theta(sort(N)) << \Theta(N)$

Os algoritmos projetados para operar com complexidade $\Theta(sort(N))$ ou $\Theta(scan(N))$ são denominados *I/O efficient* (eficiente para entrada/saída). Há alguns trabalhos recentes na literatura que propõem métodos para solucionar problemas relacionados a SIGs utilizando algoritmos que se enquadram nesta categoria [41; 19; 3].

2.7 Processamento de dados utilizando *GPU*s

Com o advento das *GPU*s (*Graphics Processing Units* - Unidades de Processamento Gráfico) com múltiplos núcleos e das arquiteturas de programação que facilitam o uso dos poderosos recursos de computação paralela, como a arquitetura *CUDA* [28], vários algoritmos foram adaptados para aproveitar o alto desempenho que pode ser alcançado com o uso dessas tecnologias. Exemplos de algoritmos incluem métodos para somatório de elementos de vetores [18], buscas em grafos e algoritmos de caminho mínimo [17] e métodos de ordenação [39].

Segundo Nickolls [33], desde o lançamento da arquitetura *CUDA* em 2007, foram desenvolvidos programas paralelos para uma grande variedade de aplicações como química computacional, algoritmos de busca e ordenação, modelos de física, etc.

Com o uso de processamento em *GPU* é possível aproveitar o grande número de multiprocessadores paralelos bem como a maior largura de banda para leitura e escrita de dados na memória propiciada por esse tipo de hardware [27]. Para isso, é necessário

[1] É transferido de forma paralela um bloco de cada um dos D discos, totalizando D blocos em cada transferência

paralelizar os algoritmos e implementá-los de modo que eles possam ser executados total ou parcialmente na *GPU*.

2.7.1 A arquitetura *CUDA*

As placas gráficas que suportam a tecnologia *CUDA* são compostas por vários processadores (mais especificamente, processadores do tipo *SIMD - Single Instruction Multiple Data* chamados de *Stream Multiprocessors - SMs*) que possibilitam a execução de muitas *threads* (processos leves) de forma paralela.

Segundo a NVIDIA [34], as *GPUs* podem prover maior poder de processamento do que *CPUs* porque elas são especializadas em executar tarefas paralelas que envolvam muitos cálculos, enquanto as *CPUs* são projetadas para executar tarefas que envolvam maior controle do fluxo de execução e *cache* de dados. A diferença física entre as duas arquiteturas pode ser visualizada na Figura 2.5: uma *GPU* dedica grande parte de sua área para as unidades de processamento (em verde), enquanto uma *CPU* dedica a maior parte da área do processador para o controle de execução e para *cache* de dados (em amarelo e laranja, respectivamente).

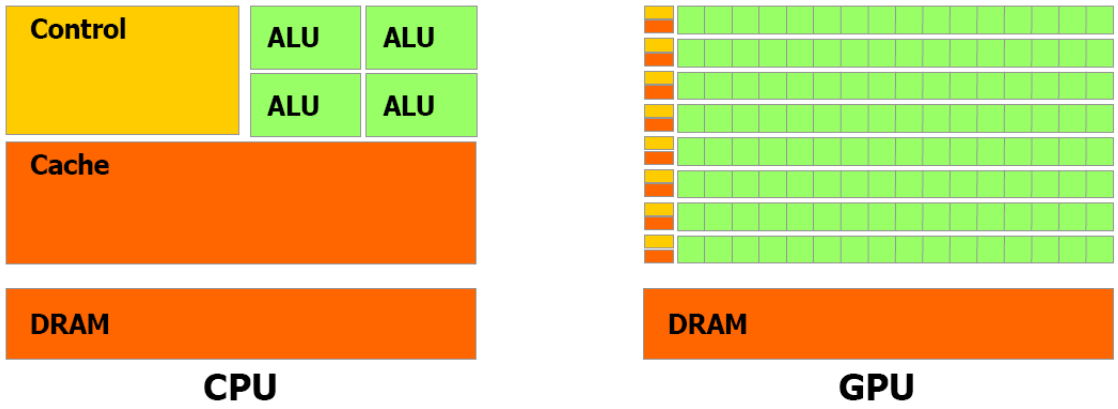


Figura 2.5. Comparação entre a estrutura da *GPU* e a estrutura de uma *CPU*. Fonte: NVIDIA [34]

Portanto, para aproveitar melhor o poder de processamento das *GPUs*, é importante que o código a ser executado nessa unidade seja composto por cálculos com paralelismo de dados, ou seja, a mesma operação deverá ser realizada em diferentes dados de forma paralela.

A arquitetura *CUDA* permite a execução paralela de trechos de código tanto na *CPU* quanto na *GPU*. Assim, o processador principal (a *CPU* ou *host*) pode executar trechos de código que envolvam menos paralelismo e código com maior quantidade de estruturas de controle de fluxo de execução (por exemplo, estruturas do tipo "if then

else") e designar para a execução na *GPU* (ou *device*) funções (chamadas de *kernels*) que possam ser aplicadas de forma paralela sobre diferentes elementos de dados.

Dessa forma, a *GPU* é utilizada como um co-processador que é capaz de executar certos tipos de tarefas de forma mais eficiente do que a *CPU*.

A arquitetura *CUDA* permite o uso de extensões à linguagem de programação C para implementar algoritmos paralelos que possam ser executados na *GPU*. Mais especificamente, o desenvolvedor define trechos de código (paralelos ou sequenciais) que são executados na *CPU* e trechos que são executados de forma paralela na *GPU* (chamados de *kernels*) [27].

Um código paralelo em *CUDA* é desenvolvido utilizando o conceito de *kernel*, que pode ser definido como uma função a ser executada na *GPU* de forma paralela sob elementos de dados diferentes (*SIMD*).

Para desenvolver aplicações eficientes em *CUDA*, é importante conhecer melhor algumas características do modelo de programação em *CUDA* como, por exemplo, a separação do código a ser executado na *CPU* e na *GPU*, os diferentes tipos de memória disponíveis nas placas gráficas, algumas características do *hardware* das *GPUs*, entre outras.

2.7.1.1 Modelo de programação com *CUDA*

Para que um programa posssa tirar melhor proveito capacidade de processamento das *GPUs* utilizando *CUDA*, ele deve ser formado por trechos de código sequencial (que serão executados na *CPU*) e trechos de código paralelos (que serão executados na *GPU*). Esses códigos são desenvolvidos utilizando extensões da linguagem de programação C/C++.

As principais extensões são relacionadas às definições de *kernels*. Um *kernel* é basicamente uma função da linguagem C que é executada N vezes em paralelo sob a forma de N *threads* [34].

As *threads* são organizadas em blocos (*blocks*) que, por sua vez, são organizados em uma grade (*grid*). Cada bloco de *threads* é executado em um *SM* e as *threads* de um mesmo bloco (*block*) podem se comunicar utilizando uma pequena quantidade de memória compartilhada de alta velocidade (na placa gráfica NVIDIA GTX 285, por exemplo, ela possui tamanho igual a 16KB). A Figura 2.6 ilustra essa organização.

Cada *thread* é identificada em um bloco de forma única pelo vetor *threadIdx*, que pode definir blocos com até três dimensões (utilizando os componentes *threadIdx.x*, *threadIdx.y*, *threadIdx.z*). Na Figura 2.6, os blocos exibidos são bi-dimensionais e há 4×3 *threads* em cada bloco.

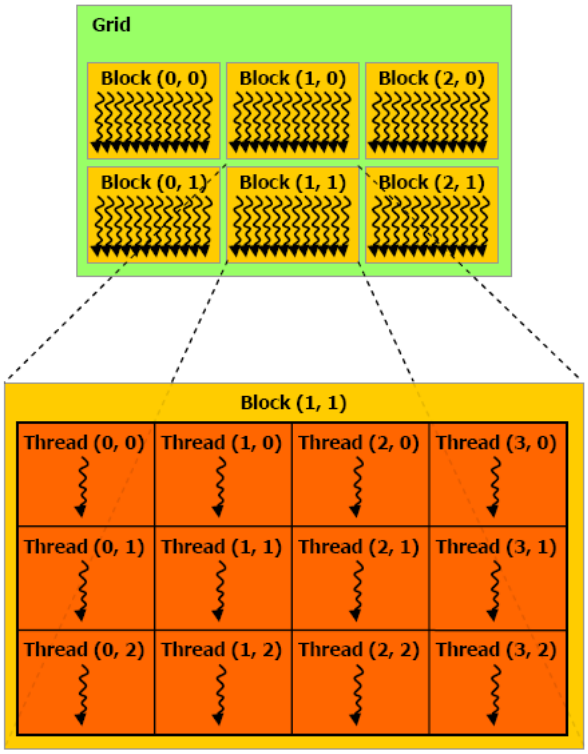


Figura 2.6. Organização de *threads* em blocos e dos blocos em uma grade: nesse exemplo há 3×2 blocos e cada bloco contém 4×3 *threads*.
Fonte: NVIDIA [34]

Além disso, uma *thread* pode identificar o bloco na qual ela está sendo executada com o uso do vetor (que pode conter uma, duas ou três dimensões) *blockIdx*. O tamanho de cada dimensão de *blockIdx* pode ser acessado pelo vetor *blockDim*.

A Figura 2.7 exibe um exemplo de *kernel* utilizado para somar duas matrizes (A e B) e armazenar o resultado em outra matriz (C). Note que, como as matrizes somadas são bi-dimensionais, as *threads* são organizadas em blocos também bi-dimensionais (onde cada bloco contém 16×16 *threads*) por motivos de simplicidade. Observe também que o *kernel SomaMatriz* é chamado utilizando a sintaxe *SomaMatriz<<< dimGrade, threadsPorBloco >>> (A,B,C)*, onde *dimGrade* é um vetor bi-dimensional que define a organização e o número (dependente do tamanho das matrizes que estão sendo somadas) de blocos utilizados e *threadsPorBloco* define a organização e o número de *threads* que serão executadas em cada bloco.

As *threads* que estão sendo executadas em um multiprocessador são agrupadas e executadas em grupos de 32 *threads* (chamados de *warp*). Esse conceito é importante porque, como cada *SM* possui a arquitetura *SIMD*, se algumas *threads* de um mesmo *warp* divergirem em um fluxo de execução (por exemplo, em um comando "if then else"), o *warp* deverá executar de forma serial cada caminho do fluxo de execução (e

```
1   __global__ void SomaMatriz(float A[N][N],float B[N][N],float C[N][N]) {
2      int i = blockIdx.x * blockDim.x + threadIdx.x;
3      int j = blockIdx.y * blockDim.y + threadIdx.y;
4
5      if (i< N && j<N)
6         C[i][j] = A[i][j] + B[i][j];
7   }
8
9   int main() {
10     ...
11     dim3 threadsPorBloco(16, 16);
12     dim3 dimGrade(N/threadsPorBloco.x,N/threadsPorBloco.y);
13     SomaMatriz<<<dimGrade,threadsPorBloco>>>(A,B,C);
14     ...
15  }
```

Figura 2.7. Exemplo de *kernel* utilizado para somar duas matrizes.
Adaptado de: NVIDIA [34]

desabilitar, em cada caminho, as *threads* que não devem ser executadas nesse caminho). Por exemplo, se 10 *threads* executarem o "if" e 22 executarem o "else", primeiro as 10 *threads* serão executadas e, depois, as 22 restantes. Isso faz com que o uso excessivo de estruturas de controle possa causar uma perda de desempenho no programa.

Outra característica importante do modelo de programação *CUDA* é a hierarquia de memória da placa gráfica. Segundo a NVIDIA [34], as *threads* podem acessar dados em diferentes tipos de memória: memória local, memória compartilhada, memória global, memória de constantes e de textura.

A memória local é acessível apenas à *thread* que a declara e é alocada em um espaço da memória global. Esse tipo de memória é lenta (assim como a global) e só é utilizada em alguns casos especiais (por exemplo, quando os registradores disponíveis para uma *thread* se esgotam).

A memória compartilhada, por sua vez, é uma zona de memória de alta velocidade que é implementada em cada *SM*. Essa área de memória é compartilhada por todas as *threads* de um mesmo bloco e pode ser acessada de forma paralela por essas *threads*.

Já a memória global, a memória de constante e a de textura são localizadas na zona de memória da placa gráfica, e são os tipos mais lentos de memória. Apesar do acesso mais lento, esses tipos de memória são utilizadas devido ao fato de estarem disponíveis em maior quantidade (1 GB ou mais de memória).

Visto que a memória global é o tipo mais lento de memória, é importante que as aplicações que utilizem *CUDA* evitem o acesso a essa memória. Isso é feito copiando-se pequenos blocos de dados da memória global para a memória compartilhada, processando esses blocos na memória compartilhada e, por fim, armazenando os resultados obtidos de volta na memória global.

2.7.1.2 A biblioteca *Thrust*

Como alguns algoritmos são de uso bastante frequente (por exemplo, algoritmos de ordenação) existem algumas bibliotecas que disponibilizam implementações eficientes desses algoritmos. O uso dessas bibliotecas permite um desenvolvimento mais rápido de softwares confiáveis e eficientes.

Um exemplo dessas bibliotecas é a *Thrust* [20], que foi desenvolvida para prover algoritmos paralelos desenvolvidos em *CUDA*. *Thrust* disponibiliza uma interface similar à disponibilizada pela *STL* (*Standard Template Library*) [38] da linguagem C++ e, com isso, permite desenvolver aplicações mais eficientes.

Thrust provê dois *containers* similares ao *container vector* da *STL*: o *host_ vector* e o *device_ vector*. O primeiro define um vetor dinâmico que é armazenado na memória principal do computador (o *host*), enquanto o segundo define um vetor que é armazenado na memória global da *GPU* (o *device*). Com o uso do *device_ vector*, é possível declarar vetores na memória da *GPU* e copiar dados para esses vetores com facilidade. Veja um exemplo de uso desses vetores na Figura 2.8.

```
1: //Declaração de vetor na memória principal
2: thrust::host_vector<int> H(2);
3: //Inicialização do vetor
4: H[0] = 14;
5: H[1] = 20;
6: //Declaração de vetor na memória global da GPU
7: thrust::device_vector<int> D;
8: //Cópia dos dados da memória principal para a GPU
9: D = H;
```

Figura 2.8. Exemplo de container definido na biblioteca *Thrust*.

Além de *containers*, a biblioteca também disponibiliza alguns algoritmos similares aos da *STL*. Exemplos de tais algoritmos incluem ordenação de dados na *GPU* e operações de redução (utilizadas para somatório de vetores, por exemplo). A Figura 2.9 exibe um exemplo de uso da operação de redução.

```
1: //Declara um vetor na memória global da GPU
2: thrust::device_vector<int> vec(3);
3: vec[0] = 1;
4: vec[1] = 5;
5: vec[2] = 10;
6:
7: //Atribui, à variável soma, o somatório dos elementos desse vetor
8: int soma
9:  = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
```

Figura 2.9. Exemplo de algoritmo (algoritmo de redução) disponibilizado pela biblioteca *Thrust*.

Outro tipo de operação disponibilizada pela *Thrust* é a operação de soma de prefixos [18]. Esse é um tipo especial de operação de redução que recebe um vetor como parâmetro e modifica os elementos desse vetor para que o i-ésimo elemento do vetor resultante represente o somatório dos elementos que estavam entre a posição 0 e a posição i desse vetor (existem as versões inclusiva e a exclusiva desse método: a primeira considera o elemento i na soma e a outra não considera). Veja a figura 2.10.

```
1  //Declara um vetor na memória global da GPU
2  thrust::device_vector<int> vec(3);
3  vec[0] = 1;
4  vec[1] = 5;
5  vec[2] = 10;
6
7  //Realiza a soma de prefixos inclusiva  e armazena o resultado
8  //no vetor original (vec)
9  thrust::inclusive_scan(vec, vec + 3, vec);
10 //Após a operação, o vetor irá conter os elementos {1,1+5=6,6+10=16}
```

Figura 2.10. Uso do algoritmo de soma de prefixos disponibilizado pela biblioteca *Thrust*.

Capítulo 3

Materiais e métodos

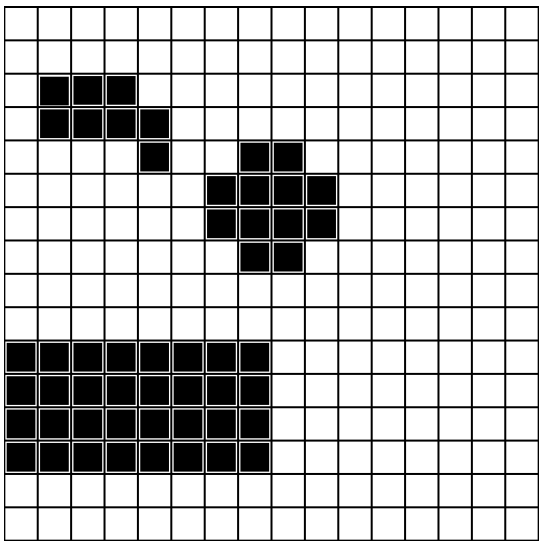
3.1 Representação de dados utilizada

Os métodos desenvolvidos nesse trabalho consideram dados de elevação representados por matrizes digitais de elevação. Mais especificamente, tais dados são armazenados em matrizes quadradas onde cada elemento representa a elevação, em metros, de uma célula correspondente do terreno.

Os *viewsheds* de pontos do terreno e os *viewsheds* acumulados são representados por matrizes binárias onde o valor 1 indica que o ponto é visível e 0 indica que o ponto não é visível e cuja dimensão é igual à dimensão dos terrenos a partir do qual a visibilidade foi calculada. Para minimizar a quantidade de memória utilizada por esses mapas, cada linha dessas matrizes é codificada utilizando bits para representar a visibilidade dos pontos (ou seja, cada conjunto de 8 pontos é codificado em um byte). Veja as Figuras 3.1(a), 3.1(b) e 3.1(c).

Além de codificar os *viewsheds* em bits, esse trabalho propõe uma forma adicional de se compactar essas matrizes: como os *viewsheds* de observadores só possuem elementos visíveis (valores 1) dentro do *raio de interesse* desses observadores, então somente a informação sobre os pontos que estão dentro desse raio precisa ser armazenada (pois os outros pontos possuem, necessariamente, valor 0). Assim, pode-se armazenar apenas um *bounding-box* quadrado com lado igual a $2R - 1$ bits. Para facilitar as operações de sobreposição dos *viewsheds*, para cada *viewshed* são armazenados todos os bytes que interceptem esse *bounding-box*. Veja as Figuras 3.2(a), 3.2(b) e 3.2(c). Note que o *viewshed* da Figura 3.2(a) poderia ser armazenado em um *bounding-box* de dimensões 1×3 (ou seja, 3 linhas com um byte em cada), mas para evitar a necessidade de operações deslocamento de bits durante a sobreposição de *viewsheds*, todos os bytes interceptados por esse *bounding-box* são armazenados (ou seja, 3 linhas com 2 bytes em cada - Figura 3.2(c)).

Esses *viewsheds* são armazenados em arquivos compostos por um cabeçalho seguido pelos bytes correspondentes à codificação do *bounding-box*. O cabeçalho descreve



(a)

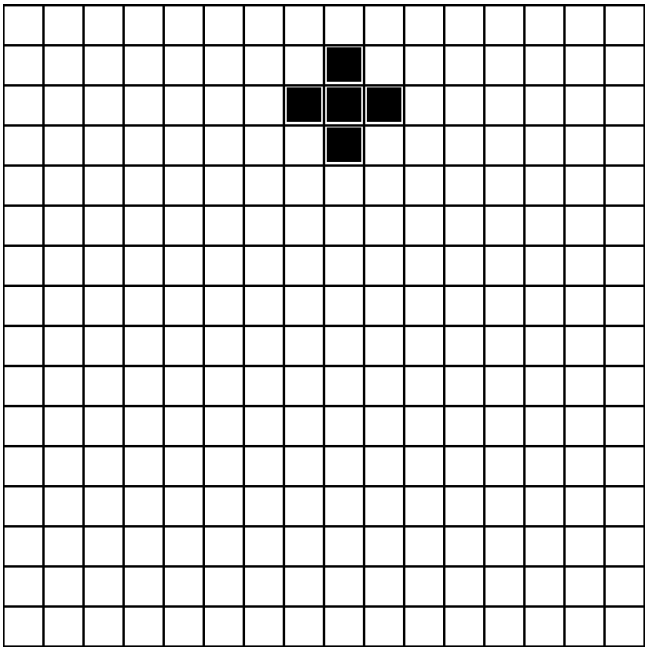
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b)

0	0
0	0
112	0
120	0
9	128
3	192
3	192
1	128
0	0
0	0
255	0
255	0
255	0
255	0
0	0
0	0

(c)

Figura 3.1. Exemplo de *viewshed* acumulado de um terreno com dimensões 16×16: (a) Visualização do *viewshed*, sendo que os pontos marcados de preto são visíveis a partir de algum observador. (b) A codificação em bits desse *viewshed*. (c) Codificação do *viewshed* em bytes.



(a)

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

(b)

0	128
1	192
0	128

(c)

Figura 3.2. Exemplo de *viewshed* de um observador: (a) Visualização do *viewshed*, sendo que os pontos marcados de preto são visíveis a partir do observador. (b) A codificação em bits do *bounding-box* desse *viewshed*. (c) Codificação do *bounding-box* do *viewshed* em bytes.

as seguintes características do *viewshed*: o número de bytes ocupados por cada linha, as coordenadas (em relação ao terreno original) do *bounding-box* que contém o *viewshed*, o raio de interesse do observador, as coordenadas do observador relativas ao *bounding-box* e as coordenadas do observador relativas ao terreno original.

3.2 Operações sobre *viewsheds*

Os métodos de posicionamento de observadores realizam diferentes operações com os *viewsheds*, sendo que as principais operações utilizadas nesse trabalho são: união do *viewshed* de um observador com um *viewshed* acumulado, cálculo de área visível em um *viewshed* (ou no *viewshed* acumulado) e cálculo de área de contribuição de um *viewshed* em relação a um *viewshed* acumulado.

A união de um *viewshed* V com um *viewshed* acumulado U é feita alinhando-se

os bytes de V com U (ou seja, posicionando a matriz V na posição correta de U) e, então, realizando-se a operação OR dos bytes de V com os bytes correspondentes de U. A forma como os bytes de V são alinhados é definida com base no cabeçalho de V.

Já o cálculo da área visível de um *viewshed* V é realizado contando-se o número de bits 1 contidos em V. Por fim, o cálculo da área de contribuição de um *viewshed* V em relação a um *viewshed* acumulado U é feito alinhando-se os bytes de V com os bytes correspondentes de U e, então, somando-se o número de bits 1 contidos na contribuição de cada byte v de V, onde a contribuição de v é definida pela fórmula

$$((u\ OR\ v)\ XOR\ u)\ =\ v\bar{u}$$

e u é o byte de U que está na posição correspondente (alinhada) a v.

3.3 Estratégia geral para posicionamento de observadores

Os métodos de posicionamento de observadores propostos neste trabalho foram baseados no método *Site* [15; 13]. Como afirmado na Seção 2.4, o método *Site* utiliza 4 etapas básicas para realizar o posicionamento de observadores. Esta seção irá descrever em maiores detalhes cada uma dessas etapas.

Por motivos de eficiência, é importante não considerar todos os pontos do terreno como candidatos a receberem observadores. Assim, o método *Site* utiliza o índice de visibilidade aproximado dos pontos para realizar uma seleção de pontos candidatos a observadores. Essa seleção ocorre nas duas primeiras etapas do método.

Na primeira etapa, o índice de visibilidade aproximado de todos os pontos do terreno é calculado. O nível de aproximação do *vix* é definido pelo usuário.

Já na segunda etapa, o conjunto de pontos candidatos P é gerado selecionando-se os pontos do terreno com maiores índices de visibilidade. Como os pontos com altos índices de visibilidade podem estar todos concentrados em uma pequena área (por exemplo, em uma região plana com montanhas em volta), o terreno é dividido em várias regiões retangulares regularmente espaçadas e, então, pontos com os maiores *vix* de cada uma dessas regiões são utilizados para formar P.

A terceira etapa do método consiste no cálculo dos mapas de visibilidade dos pontos candidatos (P). Para isso, é utilizado um método de cálculo de *viewshed* descrito em [13] que extrai o *viewshed* de um ponto do terreno e o armazena em matrizes de bits.

Por fim, a quarta etapa utiliza um método guloso para realizar o posicionamento dos pontos candidatos a observadores (os pontos de P).

Para desenvolver os métodos de posicionamento de observadores propostos neste trabalho, o método *Site* foi modificado para abordar as diferentes situações de processamento apresentadas na Seção 1.2. As seções 3.4, 3.5 e 3.6 descreverão as técnicas que foram propostas para cada uma dessas situações.

3.4 Posicionamento de observadores em terrenos armazenados em memória interna

As diferentes heurísticas de otimização combinatória para a solução do problema de posicionamento de observadores em memória interna foram desenvolvidas utilizando a estrutura original do método *Site*. Mais especificamente, as três primeiras etapas do *Site* foram mantidas com o objetivo de gerar o conjunto de pontos candidatos a receberem observadores e os respectivos *viewsheds* desses pontos. O fluxograma da Figura 3.3 destaca a etapa do método *Site* onde as heurísticas propostas foram adaptadas.

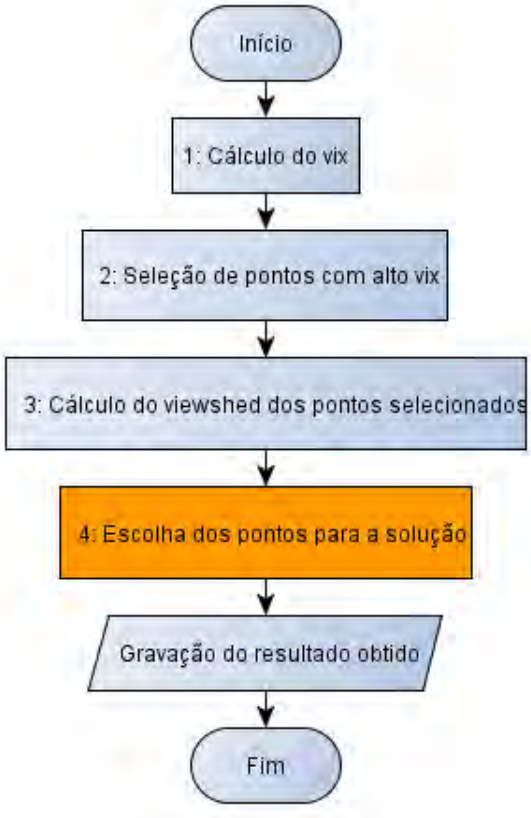


Figura 3.3. Fluxograma do método *Site* com destaque para a etapa de escolha de pontos para a solução final.

Assim, as heurísticas propostas nesta seção foram projetadas para utilizar o conjunto P de pontos candidatos a observadores (obtidos através da seleção por índice de visibilidade) e os *viewsheds* dos pontos de P. Dessa forma, o método *Site* foi modificado com o objetivo de permitir a escolha de qual heurística de posicionamento será utilizada.

3.4.1 Método guloso (*Guloso*)

Uma estratégia gulosa para o posicionamento de observadores em terrenos consiste em gerar iterativamente uma solução S, sendo que, em cada iteração, o observador P_i de P que mais contribuir com o índice de visibilidade acumulado de S é adicionado à solução corrente. Esse processo é executado até que o índice de visibilidade de S seja maior ou igual a VIX_{min}. Veja o Algoritmo 1. Essa é a estratégia original do método *Site* e será utilizada para comparação com as heurísticas propostas nesse trabalho.

Algoritmo 1 Algoritmo guloso aplicado ao problema de posicionamento de observadores

```
S ← {}
while VIX(S) < VIX_min do
  id ← 1 //Suponha que o primeiro elemento de S seja o melhor ponto.
  for j = 1 to |P| do //Encontre o ponto de P que mais contribua com o vix de S.
    if VIX(S + P_j) > VIX(S + P_id) then
      id ← j
    end if
  end for
  S ← S + {P_id}
end while
return S //Retorna a solução final.
```

3.4.2 Melhoria da área visível utilizando buscas locais

Dada uma solução S, um método de busca local gera uma vizinhança para S formada por outras soluções "próximas" a S e se movimenta por essa vizinhança com o objetivo de encontrar soluções "melhores" do que S.

O Algoritmo 2 apresenta um exemplo genérico do método de busca local. Note que o método é executado de forma iterativa substituindo, a cada iteração, a solução corrente por uma solução "vizinha" cuja função objetivo seja melhor. A busca local é interrompida quando uma solução $S*$ que não possui vizinhos "melhores" é atingida ($S*$ é chamada de ótimo local).

Algoritmo 2 Algoritmo de busca local

```
S ← SolucaoInicial
while naoOtimo(S) do //Enquanto S não for ótima localmente
  S ← vizinhoMelhor(S) //S recebe um vizinho melhor que S.
end while
return S
```

Neste trabalho, as buscas locais foram utilizadas com o objetivo de maximizar o índice de visibilidade de um número fixo de observadores. Mais especificamente, dada uma solução inicial S, o objetivo da busca local é melhorar S de modo que a solução encontrada pela busca maximize o índice de visibilidade da solução sem modificar o número de observadores utilizados na solução. Assim, dada uma solução S, uma solução vizinha S' é obtida trocando um observador de S por outro observador que não está em S. Veja a Figura 3.4. Note que essa vizinhança não modifica o valor da função objetivo de uma solução para o problema que está sendo estudado (visto que a função objetivo considera somente o número de observadores utilizados) mas, por outro lado, ela pode melhorar o índice de visibilidade da solução pois pode-se verificar este índice em várias soluções com o mesmo número de observadores. Consequentemente, ela pode ser utilizada para tornar viáveis soluções com um menor número de observadores (veja as seções 3.4.3 e 3.4.4).

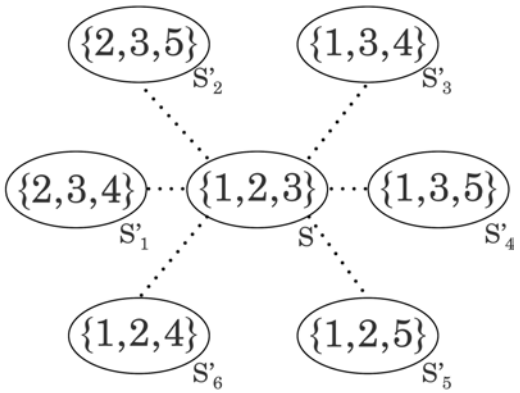


Figura 3.4. Vizinhança de uma solução $S = \{1, 2, 3\}$ gerada a partir do conjunto $P = \{1, 2, 3, 4, 5\}$

Durante a implementação da busca local, pode-se utilizar diferentes estratégias para a geração de vizinhos. Neste trabalho, foram implementadas duas versões de buscas locais utilizando as seguintes estratégias: geração de todos os vizinhos (essa estratégia é chamada de *busca local com toda vizinhança*) e geração de vizinhos considerando a localização espacial dos pontos (essa estratégia é chamada de *busca local*

espacial).

Na *busca local com toda vizinhança* geram-se todos os vizinhos da solução S atual e, então, percorre-se essa vizinhança avaliando-se o índice de visibilidade de cada vizinho S'. Ao encontrar um vizinho S' cujo *vix* seja maior do que o de S, atribui-se S' a S (ou seja, torna S' a solução atual). Esse processo é repetido até se encontrar um ótimo local.

Visto que a avaliação do índice de visibilidade de uma solução é uma operação de alto custo computacional (já que envolve a união de vários *viewsheds*), é importante desenvolver maneiras de diminuir o número de vizinhos avaliados pela busca local sem reduzir muito a qualidade das soluções obtidas.

Como a natureza do problema de posicionamento de observadores envolve a localização espacial, uma idéia para diminuir o número de soluções avaliadas consiste em se avaliar uma solução S' vizinha a S somente se o observador que está entrando na solução S e o observador que está saindo de S estiverem separados por uma distância menor do que um valor limite R_t chamado de *raio de troca*. Ou seja, durante a geração de vizinhos de S, seja P_i um ponto que está em S e seja P_j um ponto que não está em S. P_i somente é substituído por P_j caso a distância entre P_i e P_j seja menor do que R_t. Veja a Figura 3.5, que representa uma solução S: os pontos representados por triângulos e pelo pentágono pertencem à solução atual enquanto os pontos representados por quadrados e círculos não pertencem. Nesse exemplo de geração de vizinhança, o ponto representado pelo pentágono está sendo retirado e será substituído pelos pontos representados por quadrados, ou seja, na vizinhança correspondente à remoção do pentágono, somente 4 vizinhos diferentes serão avaliados (cada um conterá um dos pontos representados por quadrados). Note que a figura só representa a vizinhança correspondente à remoção do pentágono (também serão avaliados vizinhos correspondentes à remoção dos outros pontos utilizados, que são representados por triângulos).

Assim, a *busca local espacial* é executada de forma similar à *busca local com toda vizinhança*, com exceção de que a primeira avalia toda a vizinhança de uma dada solução enquanto a segunda avalia somente vizinhos "próximos". Note que se R_t for grande (maior ou igual à diagonal do terreno) a *busca local espacial* equivale à *busca local com toda vizinhança*.

3.4.3 Método *Guloso* melhorado com Busca Local (*GulosoBL* e *GulosoBL_ESP*)

Visto que os métodos de busca local descritos na Seção 3.4.2 melhoram a área coberta por uma solução S sem modificar o número de observadores utilizados na solução, foi

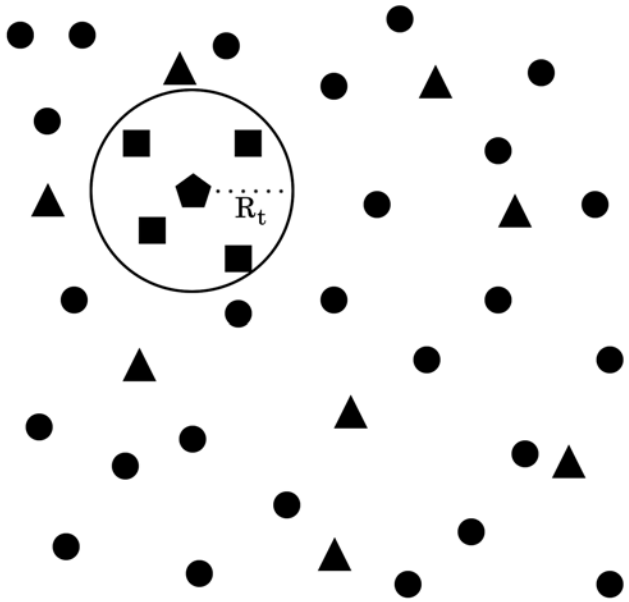


Figura 3.5. Exemplo de vizinhança considerando o componente espacial: os pontos representados por triângulos e pelo pentágono pertencem à solução atual enquanto os pontos representados por quadrados e círculos não pertencem.

proposta uma estratégia (chamada de método *GulosoBL*) para melhorar as soluções obtidas pelo método *Guloso* utilizando, em cada iteração do método, uma busca local para que o índice de visibilidade da solução atual seja melhorado. Assim, o índice de visibilidade desejado pode ser atingido realizando menos iterações e, consequentemente, um número menor de observadores.

Uma outra modificação foi realizada com o intuito de tentar reduzir o número de observadores utilizados. Mais precisamente, após a geração da solução, utiliza-se uma iteração que, a cada passo, remove de forma gulosa o observador que menos contribui para o índice de visibilidade da solução (para se determinar esse observador é necessário reconstruir soluções desconsiderando cada um dos observadores utilizados) e, então, realiza uma busca local para melhorar o *vix* da solução. Essa iteração é executada enquanto a solução atual for viável (ou seja, enquanto $VIX(S) \geq VIX_{min}$). Veja o Algoritmo 3.

A busca local utilizada na implementação do método *GulosoBL* considera toda a vizinhança das soluções. Foi implementada também uma versão do método, chamada de *GulosoBL_ESP*, que utiliza a *busca local espacial*.

A busca local do método *GulosoBL_ESP* utiliza como *raio de troca* um valor igual a três vezes o *raio de interesse* do observador. Esse valor foi escolhido baseado

Algoritmo 3 Método *GulosoBL*

$S \leftarrow \{\}$
while $VIX(S) < VIX_{min}$ **do**
  $id \leftarrow 1$ //Suponha que o primeiro elemento de S seja o melhor ponto.
  for $j = 1$ to $|P|$ **do** //Encontre o ponto de P que adiciona mais área visível a S.
    if $VIX(S + P_j) > VIX(S + P_{id})$ **then**
      $id \leftarrow j$
    end if
  end for
  $S \leftarrow S + \{P_{id}\}$
  $S \leftarrow buscaLocal(S)$ //Aplica uma busca local em O para maximizar $VIX(S)$.
end while
while $VIX(S) \geq VIX_{min}$ **do** //Enquanto S for viável
  $S* \leftarrow S$ //$S*$ armazenará a melhor solução
  $removePiorObservador(S)$ //Remova o "pior" ponto de S.
  $S \leftarrow buscaLocal(S)$ //Melhora $VIX(S)$ utilizando uma busca local
end while
return $S*$

em testes preliminares que sugerem que esse valor apresenta uma boa relação entre qualidade de solução e tempo de processamento. Vale mencionar que essa escolha deve ser confirmada através de estudos mais detalhados.

3.4.4 *GRASP*

O método *GRASP*, originalmente proposto por Resende e outros [36], consiste em criar uma solução inicial com um método construtivo e, então, melhorar essa solução utilizando um método de busca local. O método construtivo utilizado pelo *GRASP* possui um fator de aleatoriedade para que a cada execução seja gerada uma solução um pouco diferente. Esse processo de construção e melhora é repetido várias vezes e, ao final, a melhor solução encontrada é retornada pelo método. O critério de parada do método pode ser definido utilizando-se um limite de tempo, um limite para o número de iterações, entre outros. Veja o Algoritmo 4.

Neste trabalho, foi implementada uma variação do *GRASP* que consiste em utilizar um método guloso aleatorizado como método construtivo e em substituir a busca local por uma heurística que combina uma busca local com uma remoção gulosa de observadores similar à descrita na Seção 3.4.3. Como o método executa várias iterações, é importante que a busca local seja rápida e, portanto, o *GRASP* utiliza uma implementação da *busca local espacial* que, assim como no caso do método *GulosoBL_ESP*, utiliza como *raio de troca* um valor igual a três vezes o *raio de interesse* dos observa-

Algoritmo 4 Método GRASP básico

```
//Sejam: solConstrAleat(): função que gera uma solução construtiva de forma aleatória
//buscaLocal(S): função que realiza uma busca local na solução S e retorna
//a melhor solução encontrada
S* ← NULO //S* armazenará a melhor solução
while Condição de parada não for satisfeita do
  S ← solConstrAleat() //Gere uma solução construtiva de forma aleatória.
  S ← buscaLocal(S)
  if S* = NULO or f(S) > f(S*) then
    S* ← S
  end if
end while
return S*
```

dores.

Para se gerarem soluções gulosas aleatorizadas, utiliza-se a seguinte variação do método *Guloso* descrito na Seção 3.4.1: em cada iteração do método *Guloso*, seleciona-se o k-ésimo melhor ponto a ser adicionado onde k é um valor sorteado aleatoriamente entre 1 e $FatAleat$ sendo que $FatAleat$ é um fator de aleatoriedade definido pelo usuário (se $FatAleat$ for igual a 1, o método se resume ao método guloso tradicional e, quanto maior o valor de $FatAleat$, mais aleatórias são as soluções geradas).

O método *GRASP* implementado neste trabalho utilizou um valor de $FatAleat$ igual a 10. Esse valor foi escolhido com base em testes preliminares que sugerem que o uso desse valor gera soluções com boa qualidade. Vale mencionar que essa escolha deve ser confirmada através de estudos mais detalhados.

3.5 Uso da *GPU* para melhorar o desempenho do posicionamento de observadores

Como afirmado na Seção 1.2, um dos objetivos deste trabalho é adaptar as heurísticas de posicionamento de observadores propostas para que elas aproveitem o poder de processamento das *GPUs* e, consequentemente, possam ser executadas de forma mais eficiente.

Para isso, foram realizados testes de desempenho com o objetivo de identificar quais as operações realizadas pelo método de posicionamento demandam maior tempo de processamento e cuja otimização produziriam maior impacto na eficiência do método. Esses testes foram realizados utilizando a ferramenta *Gprof* de *profiler*.

Método	Área de Contribuição	Área de *viewshed* acumulado	União de *viewsheds*	Total
Guloso	88,3%	1,7%	1,8%	91,8%
GulosoBL	95.3%	0,9%	2,3%	98,5%
GulosoBL_ESP	71,0%	5,3%	13,7%	90,0%
GRASP	61,9%	6,2%	20,6%	88,7%

Tabela 3.1. Porcentagem do tempo de processamento total que foi utilizada pelas principais operações realizadas pelos métodos de posicionamento de observadores.

A Tabela 3.1 exibe os resultados do teste de *profiler* realizado com os diferentes métodos implementados. O método *GulosoBL* foi implementado utilizando a busca local tradicional (que gera todos os vizinhos), enquanto os métodos *GulosoBL_ESP* e *GRASP* utilizam o método de busca local espacial. As operações avaliadas foram as operações de cálculo de área de contribuição de um *viewshed*, cálculo da área de um *viewshed* acumulado e união de *viewsheds*.

Visto que a maior parte (em média, 92,3%) do tempo de processamento dos métodos descritos é utilizada pelas operações com os *viewsheds* (união de *viewsheds*, cálculo da área visível de um *viewshed* acumulado e cálculo de área de contribuição de um *viewshed*), as otimizações propostas foram desenvolvidas visando melhorar a eficiência dessas operações.

A estratégia utilizada para realizar o processamento em *GPU* consiste em copiar para a memória global da *GPU* o *viewshed* de todos os observadores de S e o *viewshed* acumulado do terreno (que inicialmente é vazio) e, então, realizar na própia *GPU* todas as operações que envolverem esses *viewsheds*. Assim, evita-se a transferência de um grande volume de dados entre a memória principal e a *GPU* durante cada operação (já que os dados necessários para os cálculos já estarão na *GPU*). Dessa forma, a transferência de dados entre a *GPU* e a memória da *CPU* será feita apenas para retornar os resultados das operações (por exemplo, para retornar o valor da área calculada de um *viewshed*).

O fluxograma da Figura 3.6 ilustra a estratégia de posicionamento utilizada: todas operações realizadas nas fases 1, 2 e 3 são feitas na *CPU*. Na fase 4, os *viewsheds* dos observadores são copiados para a memória da *GPU* e, por fim, na fase 5 os observadores são posicionados utilizando alguma heurística similar às descritas na Seção 3.4. A heurística de posicionamento é executada na *CPU* enquanto as operações (união, cálculo de área visível e cálculo de área adicionada) com as matrizes que representam os *viewsheds* são executadas na *GPU*.

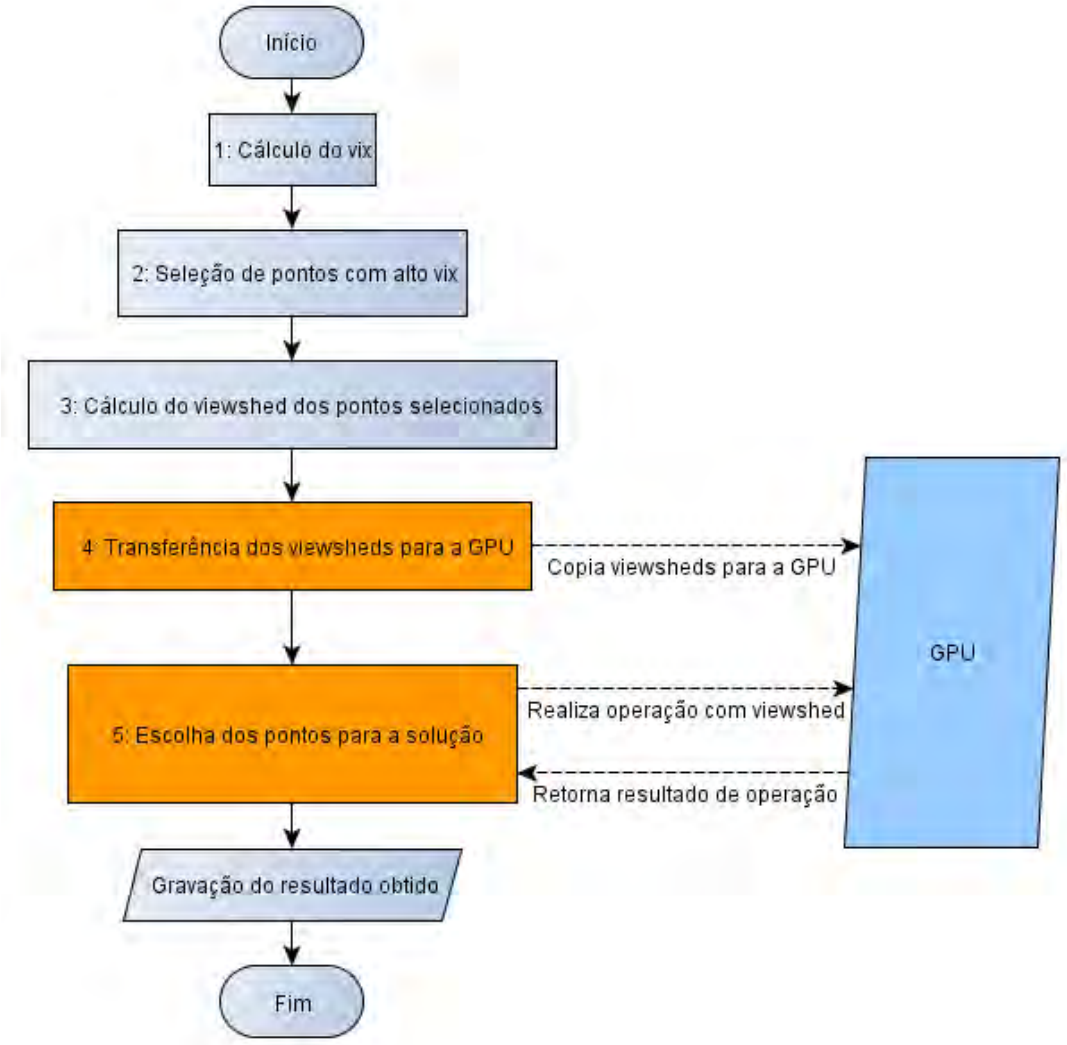


Figura 3.6. Fluxograma com a sequencia de passos utilizados para posicionar observadores em terrenos utilizando *GPU*.

Para facilitar o uso de certas operações, tanto o *viewshed* acumulado quanto os *viewsheds* dos observadores foram linearizados. Essa linearização consiste em armazenar sequencialmente em um vetor os elementos de cada linha do *viewshed* (ou seja, os elementos da segunda linha são armazenados após os da primeira linha e assim sucessivamente).

Nas Subseções 3.5.1 , 3.5.2 e 3.5.3, é descrita a forma como cada uma das operações sobre *viewsheds* foram implementadas em *CUDA*. Após a implementação dessas operações, elas foram utilizadas para substituir as operações correspondentes (que eram executadas em *CPU*) utilizadas em cada heurística descrita na Seção 3.4.

```
1  struct areaFunctor : public thrust::unary_function<int,char>
2  {
3
4      __host__ __device__
5      int operator()(char x)
6      {
7          return ((x&1)!=0) + ((x&2)!=0) +((x&4)!=0) +((x&8)!=0)
8            +((x&16)!=0) +((x&32)!=0) + ((x&64)!=0) + ((x&128)!=0);
9      }
10 };
11
12 int Viewshed::area() const { //Retorna a área visível pelo Viewshed
13                   return thrust::transform_reduce(d_vs.begin(),
14                       d_vs.end(), areaFunctor() , 0, thrust::plus<int>() );
15 }
```

Figura 3.7. Cálculo da área de um *viewshed* na *GPU*.

3.5.1 Cálculo da área de um *viewshed*

Como afirmado na Seção 3.2, o cálculo da área visível de um *viewshed* é realizado na *CPU* varrendo-se todos os pontos do *viewshed* ("codificados" em bytes) e, então, contando-se a quantidade de bits 1 que há nesses bytes. A implementação dessa funcionalidade na *GPU* foi realizada utilizando a função de redução da biblioteca *Thrust*, que implementa a operação de somatório em paralelo.

A Figura 3.7 exibe o código fonte utilizado nessa implementação. O *functor* *areaFunctor* definido na linha 1 recebe como parâmetro um byte (representado por um char) e retorna o número de bits 1 contidos nesse byte. A função de redução (chamada na linha 13) irá calcular de forma paralela o somatório do valor retornado pela aplicação desse *functor* (o terceiro parâmetro da chamada da função) em cada byte do *viewshed* em questão (representado pelo vetor *d_vs - device viewshed*, que é armazenado na memória global da *GPU*).

3.5.2 União do *viewshed* de um observador com o *viewshed* acumulado

Como a operação de união do *viewshed* de um observador com o *viewshed* acumulado é realizada através da aplicação da operação *OR* entre os elementos (bytes) do *viewshed* do observador com os elementos correspondentes do *viewshed* acumulado, essa operação foi paralelizada criando-se um *kernel* em *CUDA* que aplica a função *OR* em cada um desses elementos. Assim, para cada elemento (byte) *b* do *viewshed* é criada uma *thread* que irá aplicar a função *OR* entre esse *b* e o elemento correspondente do *viewshed* acumulado do terreno.

Veja a Figura 3.8 onde o *kernel* definido na linha 2 é utilizado para aplicar a

```
1  //Kernel que une um byte de um viewshed com o byte correspondente do viewshed acumulado do terreno
2  __global__ void uniaoByteViewshedTerreno(char *terreno, char *viewshed ,  int nLinhasTerreno,
3          int nColunasTerreno,int nLinhasViewshed,int nColunasViewshed,
4          int xMinViewshed, int yMinViewshed, int xMaxViewshed, int yMaxViewshed)
5  {
6     int xV = blockIdx.x * blockDim.x + threadIdx.x ; //Coordenada "x" no viewshed
7     int yV = blockIdx.y * blockDim.y + threadIdx.y ; //Coordenada "y" no viewshed
8     int xT = xV + xMinViewshed; //Coordenada "x" correspondente no terreno
9     int yT = yV + yMinViewshed; //Coordenada "y" correspondente no terreno
10
11
12    if ( yT>=0 && xT>=0 && yT <= yMaxViewshed && xT <= xMaxViewshed  &&
13                           yT < nLinhasTerreno && xT < nColunasTerreno) {
14        //As coordenadas do terreno e do viewshed são linearizadas
15        terreno[  (yT)*nColunasTerreno + xT   ] |=  viewshed[   (yV)*nColunasViewshed + xV ];
16    }
17 }
18
19 //Une o viewshed do observador numero "idObservador" com o viewshed
20 //acumulado do terreno
21 void Observadores::Uniao(Viewshed &terreno, int idObservador) {
22    Viewshed &vs = *viewsheds[idObservador]; //Dados do viewshed que se deseja unir
23    assert(terreno.box.getNrows()==vs.originalTerrain.nrows);
24
25    //Numero de linhas do viewshed acumulado do terreno
26    int nLinhasTerreno=vs.originalTerrain.nrows;
27    //Numero de colunas (bytes) do viewshed acumulado do terreno
28    int nColunasTerreno=terreno.box.nBytesPerRow;
29    //Numero de linhas do bounding-box do viewshed
30    int nLinhasViewshed=viewsheds[idObservador]->box.xmax-viewsheds[idObservador]->box.xmin+1;
31    //Numero de colunas (bytes) do viewshed
32    int nColunasViewshed=viewsheds[idObservador]->box.nBytesPerRow;
33
34    dim3 threadsPorBloco(16,16);
35    dim3 dimGrade(   (nColunasViewshed+threadsPorBloco.x-1)/threadsPorBloco.x,
36                 (nLinhasViewshed+threadsPorBloco.y-1)/threadsPorBloco.y    );
37
38    //Chama o kernel "uniaoByteViewshedTerreno" para unir os bytes do
39    //viewshed com os bytes do viewshed acumulado do terreno
40    uniaoByteViewshedTerreno<<<dimGrade,threadsPorBloco>>>(
41                 thrust::raw_pointer_cast(&(terreno.d_vs[0])),
42                 thrust::raw_pointer_cast( &(viewsheds[id]->d_vs[0]) ),nLinhasTerreno,
43                 nColunasTerreno,  nLinhasViewshed, nColunasViewshed,
44                 (viewsheds[idObservador]->box.xmin/8), viewsheds[idObservador]->box.ymin,
45                 (viewsheds[idObservador]->box.xmax)/8,viewsheds[idObservador]->box.ymax );
46 }
```

Figura 3.8. Código para a união de *viewshed* com um *viewshed* acumulado utilizando *GPU*.

operação *OR* entre cada byte (que está na coordenada (xV,yV)) do *viewshed* e o byte correspondente do *viewshed* acumulado (que está na coordenada (xT,yT)). Note que as coordenadas do byte b do *viewshed* que a *thread* irá processar são definidas nas linhas 6 e 7 com base no identificador único da *thread*, enquanto as coordenadas correspondentes do terreno são definidas nas linhas 8 e 9 com base nas coordenadas de b e nas coordenadas do *bounding-box* do *viewshed*. Teoricamente todas essas *threads* podem trabalhar em paralelo mas a quantidade de *threads* que serão executadas realmente em paralelo depende do número de processadores disponíveis na *GPU*.

3.5.3 Cálculo da área adicionada por um *viewshed* de observador a um *viewshed* acumulado

O cálculo da área adicionada (área de contribuição) por um *viewshed* a um *viewshed* acumulado é realizado na *GPU* utilizando um *kernel* que calcula a contribuição de cada byte do *viewshed* e, então, somando-se essas contribuições utilizando o método de somatório paralelo disponibilizado pela biblioteca *Thrust*.

Na Figura 3.9, o *kernel* "calculaContribuicaoByteViewshedTerreno" definido na linha 3 calcula a área visível adicionada por cada byte u, com coordenadas (xV,yV) (definidas com base no identificador da *thread*), do *viewshed* em questão ao byte correspondente v (com coordenadas (xT,yT)) do *viewshed* acumulado. Essa área é calculada contando-se o número de bits 1 contidos no resultado da operação $((u\ OR\ v)\ XOR\ v)$. Por fim, o *kernel* armazena o valor da área na posição do vetor temporário "contribs" correspondente ao byte u.

Após esse kernel ser chamado de forma paralela para todos os pontos do *viewshed* (linha 47), o vetor "vetorSomaTemp" irá conter em cada um dos seus números inteiros a área de contribuição do byte correspondente do *viewshed*. Para obter a contribuição total do *viewshed*, esse vetor é somado de forma paralela utilizando a operação de soma de prefixos da biblioteca *Thrust* (linha 57) e, então, essa soma é retornada pelo método (linha 61).

Os algoritmos para processamento de dados em *GPUs* normalmente apresentam maiores ganhos de desempenho em relação à *CPU* quando há uma grande quantidade de dados a serem processados visto que nesses casos a *GPU* tende a manter todos os seus processadores trabalhando por mais tempo e, com isso, o hardware da placa gráfica é utilizado de forma mais eficiente. Por exemplo, na operação de redução desenvolvida por Harris et al [18], o ganho de desempenho da *GPU* em relação à *CPU* aumentou cerca de 200 vezes quando a quantidade de dados a serem processados subiu de 1024 para 16777216 elementos. Assim, foi implementada uma versão do método de cálculo de área de contribuição que calcula o valor dessa área para um conjunto de observadores e, portanto, processa uma maior quantidade de dados.

Na Figura 3.10, o *kernel* "calculaContribuicaoByteViewshedTerreno" é chamado (linha 37) para cada *viewshed* de número *id* que esteja no vetor $idViewsheds$ e a área de contribuição de cada byte do i-ésimo *viewshed* cuja contribuição está sendo calculada é armazenada a partir da posição $i \times tamanhoMaximoViewshed$ no vetor "vetorSomaTemp", onde $tamanhoMaximoViewshed$ indica o número máximo de bytes que um *viewshed* ocupará.

A Figura 3.11 exemplifica o uso do vetor "vetorSomaTemp": supondo que deseja-

```
1  //Kernel que calcula a área de contribuição de um byte do viewshed
2  //em relação ao byte correspondente do terreno.
3  __global__ void calculaContribuicaoByteViewshedTerreno(char *terreno, char *viewsheds,int *contribs,
4     int nLinhasTerrenos, int nColunasTerreno,int nLinhasViewshed,int nColunasViewshed,
5     int xMinViewshed,int yMinViewshed, int xMaxViewshed, int yMaxViewshed)
6  {
7      int xV = blockIdx.x * blockDim.x + threadIdx.x ; //Coordenada "x" no viewshed
8      int yV = blockIdx.y * blockDim.y + threadIdx.y ; //Coordenada "y" no viewshed
9      int xT = xV + xMinViewshed; //Coordenada "x" correspondente no terreno
10     int yT = yV + yMinViewshed; //Coordenada "y" correspondente no terreno
11
12
13     if ( yT>=0 && xT>=0 && yT <= yMaxViewshed && xT <= xMaxViewshed && yT < nLinhasTerreno
14                                                                    && xT <nColunasTerreno) {
15       int byteTerreno = terreno[  (yT)*nColunasTerreno + xT   ];
16       int byteViewshed= viewsheds[   (yV)*nColunasViewshed + xV ];
17       int c = (byteTerreno |  byteViewshed ) ^ byteTerreno ;
18
19       contribs[yV*vcolumns + xV] = ((c&1)!=0) + ((c&2)!=0) + ((c&4)!=0) + ((c&8)!=0) + ((c&16)!=0) +
20                                                          ((c&32)!=0) + ((c&64)!=0) + ((c&128)!=0);
21    }
22 }
23
24 //Calcula a área de contribuição do viewshed do observador numero "idObservador"
25 //em relação ao viewshed acumulado do terreno
26 int Observers::areaContrib(Viewshed &terreno,int idObservador) {
27    Viewshed &vs = *viewsheds[idObservador]; //Dados do viewshed em questão
28    assert(terreno.box.getNrows() == vs.originalTerrain.nrows);
29
30    int nLinhasTerreno = vs.originalTerrain.nrows; //Numero de linhas do viewshed acumulado do terreno
31    //Numero de colunas (bytes) do viewshed acumulado do terreno
32    int nColunasTerreno =  terreno.box.nBytesPerRow;
33    //Numero de linhas do bounding-box do viewshed
34    int nLinhasViewshed = viewsheds[idObservador]->box.xmax-viewsheds[idObservador]->box.xmin +1;
35    //Numero de colunas (bytes) do viewshed
36    int nColunasViewshed = viewsheds[idObservador]->box.nBytesPerRow;
37
38    //Inicializa com 0 o vetor onde serão armazenados temporariamente a contribuição de cada
39    //byte do viewshed
40    thrust::fill(vetorSomaTemp.begin(), vetorSomaTemp.begin() + nLinhasViewshed*nColunasViewshed, 0);
41
42    dim3 threadsPorBloco(16, 16);
43    dim3 dimGrade(   (nColunasViewshed+threadsPorBloco.x-1)/threadsPorBloco.x,
44                     (nLinhasViewshed+threadsPorBloco.y-1)/threadsPorBloco.y   );
45
46    //Chama o kernel para preencher o vetorSomaTemp com as contribuições de cada byte
47    calculaContribuicaoByteViewshedTerreno<<<dimGrid, threadsPerBlock>>>(
48        thrust::raw_pointer_cast(&(terreno.d_vs[0])),
49        thrust::raw_pointer_cast(&(viewsheds[idObservador]->d_vs[0])),
50        thrust::raw_pointer_cast(&vetorSomaTemp[ 0 ]), nLinhasTerreno,  nColunasTerreno,
51        nLinhasViewshed, nColunasViewshed, (viewsheds[idObservador]->box.xmin/8),
52        viewsheds[idObservador]->box.ymin,(viewsheds[idObservador]->box.xmax)/8,
53                                                  viewsheds[idObservador]->box.ymax );
54
55    //Soma a contribuição de cada byte (armazenada no vetorSomaTemp) utilizando a redução
56    //inclusive_scan da biblioteca Thrust
57    thrust::inclusive_scan(vetorSomaTemp.begin(),
58          vetorSomaTemp.begin() + nLinhasViewsheds*nColunasViewshed,vetorSomaTemp.begin() );
59    //Como a redução é do tipo inclusive_scan, a contribuição total ficará no último
60    //elemento do vetor
61    return vetorSomaTemp[nLinhasViewshed*nColunasViewshed-1];
62 }
```

Figura 3.9. Código utilizado para o cálculo da área de contribuição de um *viewshed* a um *viewshed* acumulado utilizando *GPU*.

```
1  //Kernel que calcula a área de contribuição dos "numViewsheds" viewsheds
2  //com base no vetor que contém as somas de prefixo
3  __global__ void calculaSomaContribuicao( int *resultado,int *somaPrefixo, int numViewsheds,
4                                                                              int tamVs) {
5     int id = blockIdx.x * blockDim.x + threadIdx.x ;
6     if (id >0 && id < numViewsheds) {
7        resultado[id] = somaPrefixo[ (id+1)*tamVs -1 ] - somaPrefixo[ (id)*tamVs-1 ];
8     }
9  }
10 //Calcula a área de contribuição dos viewsheds de cada observador cujos ids
11 //estão no vetor "idViewsheds".
12 //Os resultados são armazenados no vetor "contribs"
13 void Observers::areaContribObservadores(Viewshed &terreno,thrust::host_vector<int> &contribs,
14                                                    thrust::host_vector<int> &idViewsheds) {
15    int nLinhasTerreno = terreno.box.nrows; //Número de linhas do viewshed acumulado do terreno
16    //Número de colunas (bytes) do viewshed acumulado do terreno
17    int nColunasTerreno = terreno.box.nBytesPerRow
18    //Número de pontos que serao avaliados pelo método
19    int numViewshedsAvaliados = idViewsheds.size();
20    //Número máximo de bytes que um viewshed terá
21    int tamanhoMaximoViewshed = maxLinhasViewshed*maxColunasViewshed;
22    //Inicializa com 0 o vetor onde serão armazenados temporariamente a contribuição de cada
23    //byte dos viewsheds
24    thrust::fill(vetorSomaTemp.begin(), vetorSomaTemp.begin() +
25                                       numViewshedsAvaliados*tamanhoMaximoViewshed, 0);
26
27    dim3 threadsPorBloco(16, 16);
28    for(int i= 0;i<numViewshedsAvaliados;i++) {
29       int id = idViewsheds[i]; //O método calculará a contribuicao do viewshed id
30       int nLinhasViewshed = viewsheds[id]->box.xmax-viewsheds[id]->box.xmin +1;
31       int nColunasViewshed = viewsheds[id]->box.nBytesPerRow;
32       dim3 dimGrade(   (nColunasTerreno+threadsPorBloco.x-1)/threadsPorBloco.x,
33                     (nLinhasViewshed+threadsPorBloco.y-1)/threadsPorBloco.y   );
34
35       //Chama o kernel para preencher o "vetorSomaTemp" (nas posicoes correspondentes
36       //a cada viewshed "id") com as contribuicoes de cada byte do viewshed "id"
37       calculaContribuicaoByteViewshedTerreno<<<dimGrade, threadsPorBloco>>>(
38             thrust::raw_pointer_cast(&(terreno.d_vs[0])),
39             thrust::raw_pointer_cast( &(viewsheds[id]->d_vs[0]) ),
40             thrust::raw_pointer_cast(&vetorSomaTemp[i*tamanhoMaximoViewshed ]),
41             nLinhasTerreno,  nColunasTerreno,
42             nLinhasViewshed, nColunasViewshed, (viewsheds[id]->box.xmin/8),
43             viewsheds[id]->box.ymin, (viewsheds[id]->box.xmax)/8,viewsheds[id]->box.ymax );
44    }
45
46    //Soma a contribuição de cada elemento dos viewsheds utilizando a operação de soma de prefixos
47    thrust::inclusive_scan(vetorSomaTemp.begin(),
48       vetorSomaTemp.begin() + numViewshedsAvaliados*tamanhoMaximoViewshed, vetorSomaTemp.begin() );
49
50    dim3 threadsPorBloco2(512);
51    dim3 dimGrade2(   (numObservers+511)/512  );
52
53    //Chama o kernel que calcula a área de contribuição dos viewsheds do vetor "idViewsheds" com
54    //base no vetor que contém as comas de prefixo
55    calculaSomaContribuicao<<<dimGrade2,threadsPorBloco2>>>(thrust::raw_pointer_cast(&vetorResult[0]),
56          thrust::raw_pointer_cast(&vetorSomaTemp[0]), numViewshedsAvaliados, tamanhoMaximoViewshed );
57
58    //Calcula a área de contribuicao do primeiro observador do vetor "idViewsheds"
59    vetorResultado[0] = vetorSomaTemp[tamanhoMaximoViewshed -1];
60
61    //Copia os dados do vetor "vetorResultado" (que está armazenado na memória global da GPU) para
62    //o vetor contribs (que está armazenado na memória principal do computador)
63    contribs = vetorResultado;
64 }
```

Figura 3.10. Código utilizado para o cálculo da área de contribuição de um conjunto de *viewsheds* a um *viewshed* acumulado utilizando *GPU*.

	Viewshed 1				Viewshed 2				Viewshed 3			
Posição	0	1	2	3	4	5	6	7	8	9	10	11
Valor original	1	2	1	0	2	3	8	8	5	0	2	0
Soma de prefixo	1	3	4	4	6	9	17	25	30	30	32	32

Figura 3.11. Exemplo do conteúdo do vetorSomaTemp.

se calcular a área de contribuição de três *viewsheds* e que cada *viewshed* ocupa no máximo 4 bytes, a área de contribuição (linha "Valor Original") dos bytes do primeiro *viewshed* será armazenada nas posições 0, 1, 2 e 3 do vetor; a área de contribuição dos bytes do segundo *viewshed* será armazenada nas posições 4, 5, 6 e 7 e a área de contribuição dos bytes do terceiro *viewshed* será armazenada nas posições 8, 9, 10 e 11. Assim, a área de contribuição total de cada *viewshed* pode ser obtida somando-se os elementos desse vetor que são correspondentes ao *viewshed* em questão.

Como é mais eficiente realizar a soma de todos os elementos do que realizar várias somas menores [18], foi utilizada uma soma de prefixos (chamada na linha 47 - Figura 3.10) para calcular a soma de prefixos dos elementos do vetor "vetorSomaTemp" e, então, a área de contribuição total de cada *viewshed* foi obtida com base nessa soma. Note que, após o somatório de prefixos, o vetor $vetorSomaTemp[i]$ indica o somatório dos elementos $0, 1, .., i$ que estavam nesse vetor e, portanto, o somatório dos elementos que estavam entre as posições i e j desse vetor pode ser obtido pela expressão $vetorSomaTemp[j] - vetorSomaTemp[i-1]$. O *kernel* "calculaSomaContribuicao" (chamado na linha 55) é utilizado para calcular o somatório em cada intervalo de dados correspondente a um *viewshed* utilizando essa idéia.

A linha "Soma de prefixo" da Figura 3.11 ilustra a idéia de se utilizar o somatório de prefixos para calcular a área de contribuição total de cada *viewshed*: a contribuição do primeiro *viewshed* pode ser obtida na posição 3 do vetor (valor 4); a contribuição do segundo *viewshed* pode ser obtida subtraindo-se o elemento da posição 3 do elemento da posição 7 ($25 - 4 = 21$) e a contribuição do terceiro *viewshed* pode ser obtida subtraindo-se o elemento da posição 7 do elemento da posição 11 ($32 - 25 = 7$).

3.6 Posicionamento de observadores em terrenos armazenados em memória externa

3.6.1 Adaptação do método *Site*

Uma forma de se realizar o posicionamento de recursos em terrenos grandes consiste em dividir a matriz de elevação do terreno em matrizes menores que possam ser processadas

em memória interna e, então, realizar o posicionamento utilizando o *Site* nessas submatrizes.

Mais especificamente, dado um terreno T representado por uma matriz de elevação de dimensão $n \times n$, suponha que seja possível realizar, na memória interna, o posicionamento de recursos em uma matriz de dimensão $m \times m$ (sendo $m < n$). Assim a matriz T é subdividida em k submatrizes menores T_1, T_2, ... , T_k de dimensões $m \times m$ e o *Site* pode ser aplicado a cada matriz individualmente.

Essa abordagem geralmente não gera resultados satisfatórios pois, ao processar um terreno T_a, não é considerada a influência da visibilidade de observadores posicionados próximos às bordas de um terreno T_b vizinho a T_a. Ao desconsiderar essa influência, são geradas redundâncias em áreas próximas às bordas e, além disso, o algoritmo guloso pode deixar de posicionar algum observador em uma área próxima às bordas de um terreno T_j pois parte do *viewshed* desse observador estaria fora de T_j e, portanto, não adicionaria muita área visível à subdivisão atual do terreno (apesar dele poder adicionar área visível a outro terreno T_c, vizinho a T_j).

Para ilustrar este problema, veja a Figura 3.12. Nela, o terreno foi dividido em quatro terrenos T_1, T_2, T_3 e T_4 e há um observador próximo às bordas de T_2. Note que o *viewshed* desse observador influencia a visibilidade de todos os terrenos. Se o posicionamento de observadores for feita em cada terreno individualmente, apenas a visibilidade adicionada a T_2 seria considerada.

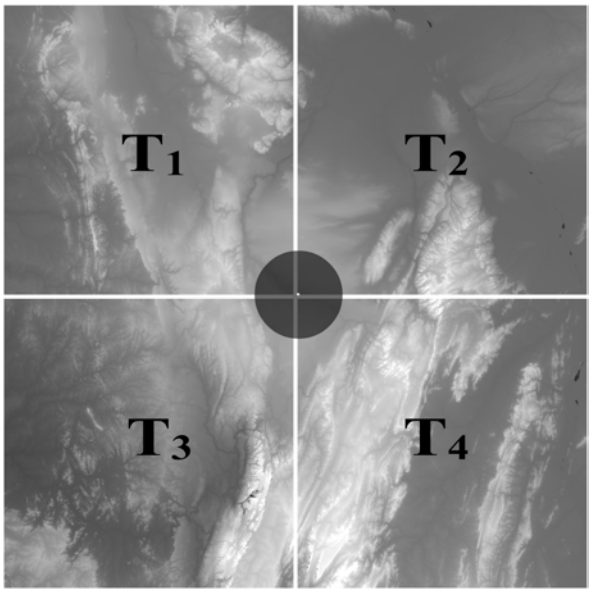


Figura 3.12. Divisão de um terreno em quatro terrenos menores

Outro inconveniente ainda mais importante dessa abordagem é que em cada sub-

divisão seria necessário atingir a taxa de cobertura desejada. Isso pode levar a situações em que a solução não pode ser obtida caso alguma subdivisão seja de difícil visibilidade.

3.6.2 O algoritmo proposto

O algoritmo proposto, chamado de *EMSite* (*External Memory Site*), estende a idéia descrita na Seção 3.6.1 para levar em consideração a contribuição de observadores posicionados próximos às bordas dos terrenos divididos.

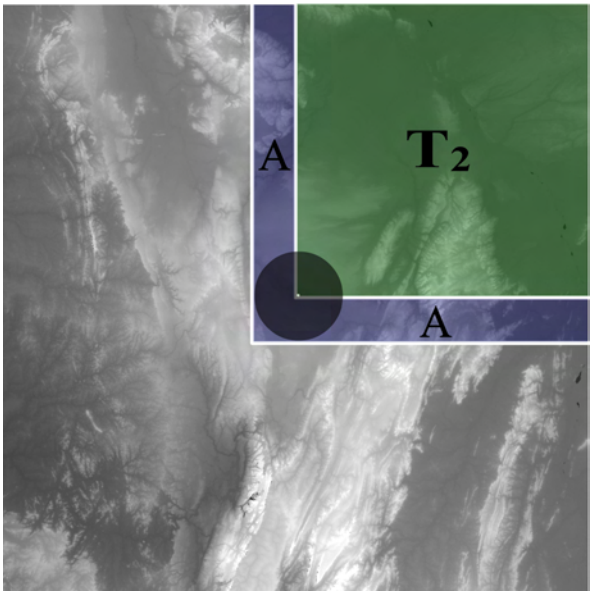


Figura 3.13. Adição de borda extra a T_2

A idéia utilizada consiste em dividir os terrenos em regiões menores que possam ser processadas em memória interna, sendo que a cada região é adicionada uma borda de largura igual ao raio de interesse (a submatriz aumentada com a borda deve possuir tamanho suficiente para processamento em memória interna). Na figura 3.13, os observadores serão posicionados apenas na área do terreno que não inclui a borda, mas a visibilidade deles é considerada tanto na região sendo processada quanto na borda. A região indicada por T_2 corresponde a uma subdivisão do terreno e a área indicada por A corresponde à borda adicionada a T_2. Note que observadores só podem ser posicionados em T_2, mas a visibilidade deles pode atingir a área A. Com isso, a visibilidade de cada observador é considerada na sua totalidade durante o posicionamento.

Para representar a visibilidade do terreno, uma matriz V é mantida em disco sendo que ela é inicializada não contendo nenhum ponto visível (situação inicial do terreno). Assim, após a subdivisão do terreno, o método *Site* é aplicado a cada região T_i (acrescido da borda correspondente) e a matriz de visibilidade gerada pelo *Site* é

sobreposta a V. Antes de processar cada região T_i, a porção de V correspondente a T_i acrescido da borda é carregada pelo *Site*, que a utilizará como estado inicial do terreno. Dessa forma, é evitada a redundância em pontos da região que está sendo processada e que já sejam visíveis a partir de observadores posicionados próximo às bordas de alguma região T_j vizinho a T_i.

Como critério de parada, o algoritmo avalia duas características da solução corrente: o número de observadores posicionados e a área de cobertura atingida. Assim, o algoritmo pára quando é atingido o número máximo de observadores definidos para cada região ou quando a área coberta do terreno é maior ou igual à área mínima.

Uma questão importante é a forma de se distribuir os observadores nos terrenos divididos. Como o *Site* é aplicado a cada subdivisão do terreno, é necessário decidir o numero de observadores que serão adicionados a cada uma dessas partes. Além disso, uma boa estratégia gulosa é realizar o posicionamento dos observadores primeiro nos terrenos onde o posicionamento gere melhores resultados.

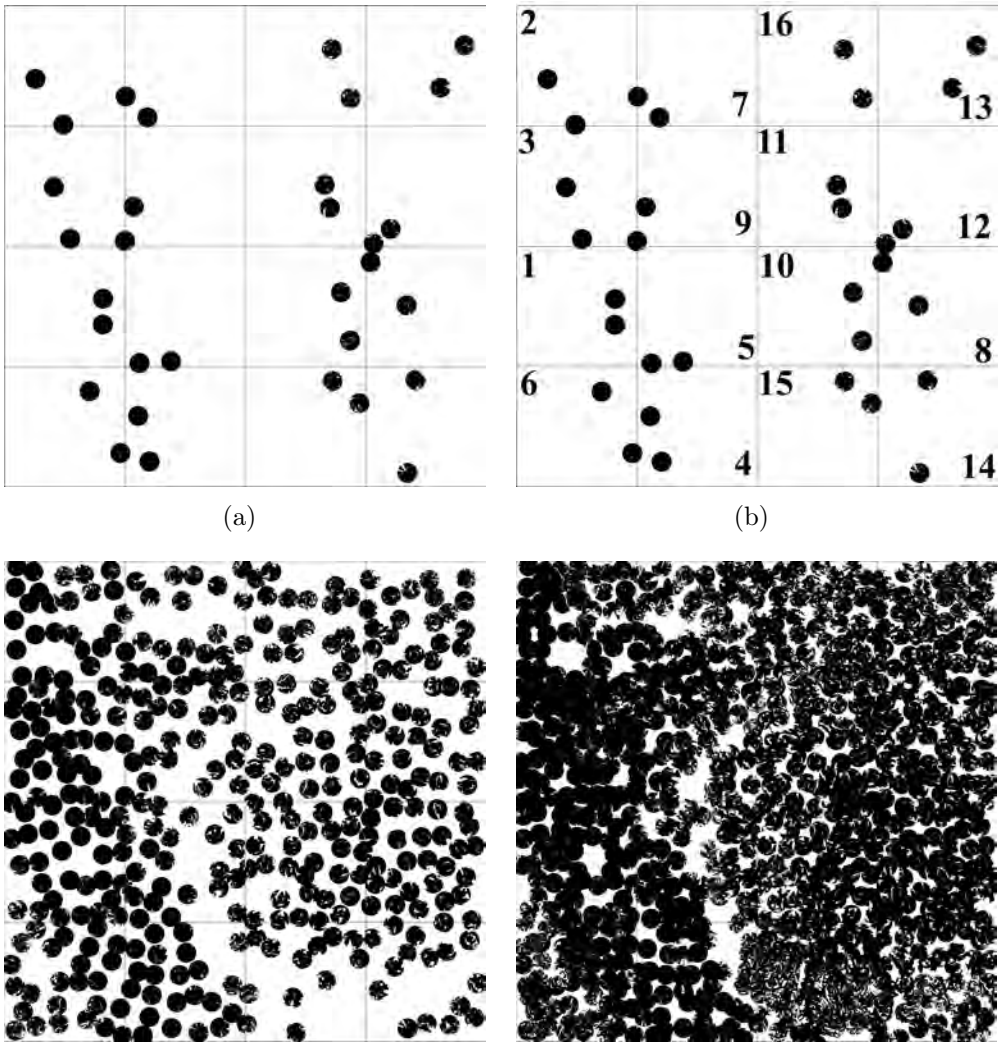
Para determinar os terrenos mais promissores, é feito inicialmente um posicionamento de k observadores utilizando o *Site* em cada subdivisão do terreno. Com essa amostragem, é definida a ordem em que essas subdivisões serão processadas, ou seja, as subdivisões onde a adição dos observadores gerou as maiores taxas de cobertura serão as primeiras a serem processadas.

A distribuição dos observadores em cada região é realizada através de uma heurística que utiliza um valor mínimo δ pré-definido para a área adicionada por um observador. Assim, para cada região, o método adiciona novos observadores enquanto a contribuição deste observador for maior do que δ.

O valor de δ é definido com base na área de um *viewshed* completo (ou seja, um *viewshed* com todos os pontos visíveis). Mais especificamente, δ é definida pelo valor da contribuição mínima que um observador deve acrescentar ao *viewshed* acumulado para poder ser adicionado à solução corrente.

Se, ao final dessa etapa, o *EMSite* não atingir o critério de parada, então o método é repetido até que esse critério seja satisfeito. Nesse caso, o valor de δ é reduzido para que o método adicione mais observadores em cada região. Assim, em cada iteração, δ é dividido por uma constante, diminuindo, assim, a exigência de área visível mínima para os observadores adicionados. Além disso, em cada iteração, a ordem de processamento das subdivisões do terreno é definida com base na quantidade de área visível adicionada pelo último observador posicionado nas subdivisões durante a iteração anterior. Assim, as subdivisões mais promissoras são as primeiras a serem processadas.

A Figura 3.14 exibe um exemplo das fases do *EMSite*. Foi utilizado um *raio de interesse* pequeno para que a quantidade de observadores posicionados seja grande. Note



(a) (b)

(c) (d)

Figura 3.14. Fases do método *EMSite* (a) Dois observadores são posicionados em cada região do terreno. (b) As regiões são ordenadas em ordem decrescente pelo incremento que o último observador posicionado nessa região adicionou à área do *viewshed* acumulado. (c) Observadores foram posicionados em cada região até que a área de contribuição do último observador seja menor do que δ. (d) O valor de δ foi reduzido e mais observadores foram posicionados até que a área de cobertura desejada fosse alcançada.

que os primeiros observadores posicionados pelo método são círculos quase completos (ou seja, representam observadores com altos índices de visibilidade) e que os últimos observadores posicionados (Figuras 3.14(c) e 3.14(d)) possuem índices de visibilidade menores (ou seja, seus *viewsheds* possuem mais áreas brancas).

Capítulo 4

Experimentos para avaliação dos métodos

Os métodos de posicionamento de observadores para processamento em *CPU* foram implementados em *C++* e compilados com o compilador g++ 4.3.4 (utilizando diretiva de compilação -O3). Já as versões com processamento em *GPU* foram implementadas em *C++/CUDA* e compiladas com o compilador nvcc 3.0 (também utilizando diretiva de compilação -O3). Todos os testes foram executados em um computador com sistema operacional Linux Ubuntu 10.04, processador Core 2 Duo de 2.9 GHz com 4 GB de memória, disco rígido Samsung de 1 TB e 7200RPM e placa gráfica com *GPU* NVIDIA GTX 285 com 1 GB de memória e 240 núcleos de processamento.

4.1 Experimentos realizados em memória interna

Os testes realizados com os métodos de posicionamento de observadores em memória interna (tanto com processamento em *GPU* quanto em *CPU*) utilizaram dois terrenos obtidos a partir da página do Nasa SRTM [30]:

- Terreno 1: Terreno com dimensões 1201×1201 células (dimensão de um bloco do SRTM3, com 90 metros de resolução) e que contém parte do estado de Minas Gerais (esse terreno inclui a cidade de Viçosa). A Figura 4.1 destaca o bloco correspondente ao terreno e a Figura 4.3(a) exibe uma visualização desse bloco gerada a partir dos dados de elevação.

- Terreno 2: Terreno com dimensões 3601×3601 células (dimensão de um bloco do SRTM1, com 30 metros de resolução) e que contém parte do estado de Nova Jersey (EUA). A Figura 4.2 destaca o bloco correspondente ao terreno e a Figura 4.3(b) exibe uma visualização desse bloco gerada a partir dos dados de elevação. Foi utilizado um terreno dos EUA devido à facilidade de se obter terrenos com maiores resoluções.

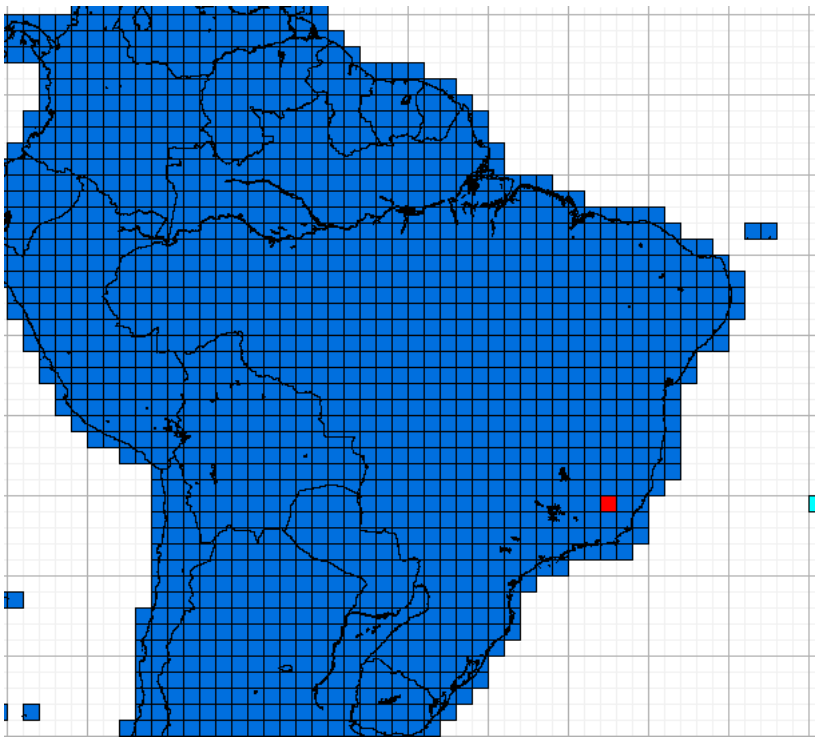


Figura 4.1. Mapa com destaque (em vermelho) para o bloco de dados referente ao Terreno 1. Adaptado de [30]

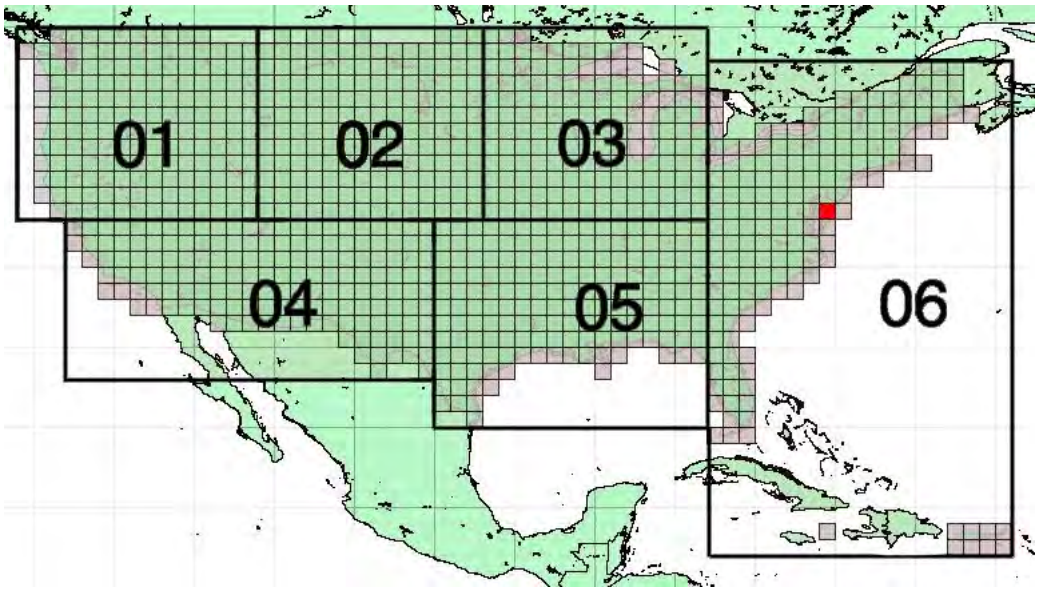
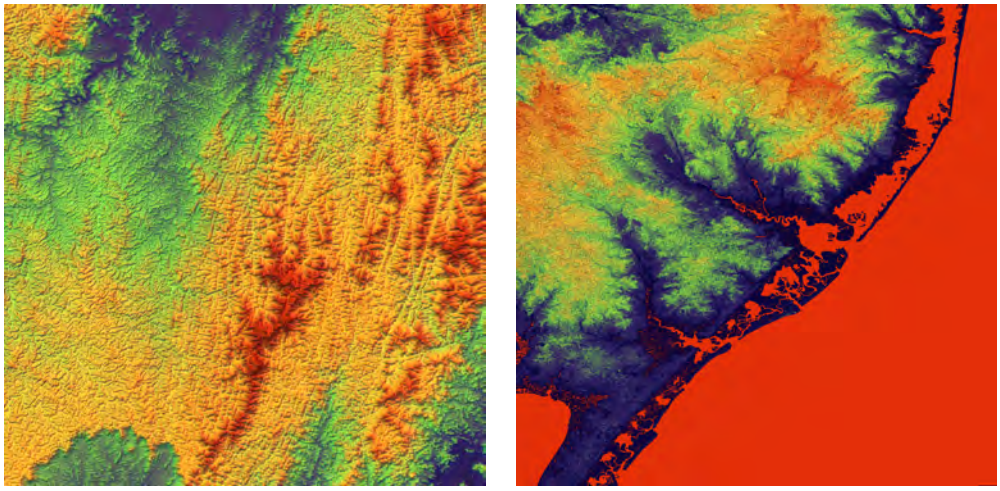


Figura 4.2. Mapa com destaque (em vermelho) para o bloco de dados referente ao Terreno 2. Adaptado de [30]



(a) (b)

Figura 4.3. Terrenos utilizados nos testes: (a) Terreno que contém parte do estado de Minas Gerais. (b) Terreno que contém parte do estado de Nova Jersey (EUA). Essas visualizações foram geradas com o programa POV-Ray [29].

Para a criação do conjunto inicial de pontos candidatos a receberem observadores foi utilizado o método de seleção por índice de visibilidade proposto por Franklin et al. [13]. Foram selecionados 1000 pontos candidatos no Terreno 1 e 3000 pontos no Terreno 2.

O *viewshed* dos pontos candidatos a receberem observadores foi calculado utilizando o método *viewshed* [13] e a elevação acima do solo desses observadores e dos alvos foi definida como 30 metros (um valor útil, por exemplo, considerando torres de telefonia celular).

Os métodos propostos foram comparados com o método *Guloso* utilizado por Franklin et al. [15; 43] e descrito na Seção 3.4.1 pois esse é o método de posicionamento de observadores a que foi possível ter acesso que é capaz de processar terrenos em alta resolução. Os métodos descritos por Deane et al. [10], por exemplo, só foram testados em terrenos contendo poucos pontos que poderiam ser visíveis pelos observadores (entre 30 e 120 pontos) e considerando poucos pontos candidatos a receberem observadores (entre 12 e 18 pontos). Já os métodos propostos por Kim et al. [21] só puderam ser executados pelos autores em terrenos de baixa resolução (terrenos de 40×40 pontos) e considerando poucos pontos como candidatos a receberem observadores (442 pontos).

A Tabela 4.1 exibe o número de observadores utilizados pelos métodos *Guloso*, *GulosoBL*, *GulosoBL_ESP* e *GRASP* para realizar o posicionamento nos terrenos 1 e 2, considerando raios de interesse com 100, 200 e 300 pontos (coluna R) e índices de visibilidade mínimos desejados equivalentes a 25%, 50%, 75%, 85% e 95% da área dos

terrenos (coluna *VIX*). O método *GRASP* utilizou como critério de parada um limite de tempo de 4200 segundos.

Para comparar o tempo de processamento das heurísticas considerando o processamento em *CPU* e *GPU*, os diferentes casos de teste descritos anteriormente foram executados utilizando as implementações das heurísticas para esses dois dispositivos. A Tabela 4.2 exibe esses tempos (em segundos). É importante mencionar que, como o tempo de processamento do método *GRASP* foi utilizado como critério de parada (o método é interrompido após ser executado por 4200 segundos) essa tabela exibe o tempo gasto pelo *GRASP* para encontrar a melhor solução (a solução final) e não o tempo total que o método ficou executando, que seria constante.

Os gráficos das Figuras 4.4, 4.5 e 4.6 exibem o tempo de execução (na *CPU*) e o número de observadores utilizados pelas heurísticas para posicionar observadores no Terreno 1 considerando raios de interesse de, respectivamente, 100, 200 e 300 pontos. Vale mencionar que os gráficos não estão na mesma escala visto que o objetivo desses gráficos é comparar as heurísticas considerando um mesmo caso de teste.

Observe que, exceto para índices de visibilidade pequenos (25% ou 50%), todas as heurísticas propostas apresentaram soluções utilizando menos observadores do que o método *Guloso*. Essas soluções utilizaram até $3,7\%$ menos observadores no caso de teste que utiliza raio de 300 pontos e índice de visibilidade desejado igual a 85%.

Note também que, nesses casos de teste, a diferença na qualidade dos resultados variou pouco entre as heurísticas propostas e que, quando o índice de visibilidade desejado chegou a 95%, houve um crescimento muito acentuado no número de observadores utilizados pelos métodos. Esse crescimento ocorreu devido a características montanhosas do terreno, que fazem com que muitos pontos possuam baixos índices de visibilidade. Como o aumento no número de observadores utilizados por cada heurística cresceu de forma similar, a diferença de qualidade apresentada por elas foi menor nos casos de teste cujos índices de visibilidade desejados são altos.

Devido à sua simplicidade, o método *Guloso* apresentou o menor tempo de processamento em todos os casos de teste. Dentre as heurísticas propostas os métodos *GRASP* e *GulosoBL_ESP* obtiveram, em geral, menores tempos de execução. Como esperado, isso sugere que a *busca local espacial* utilizada por esses dois métodos apresenta um desempenho melhor do que a *busca local com toda a vizinhança*, que é utilizada no método *GulosoBL*.

As Figuras 4.7, 4.8 e 4.9 exibem gráficos com o número de observadores utilizados e o tempo de execução (na *CPU*) das heurísticas considerando o Terreno 2.

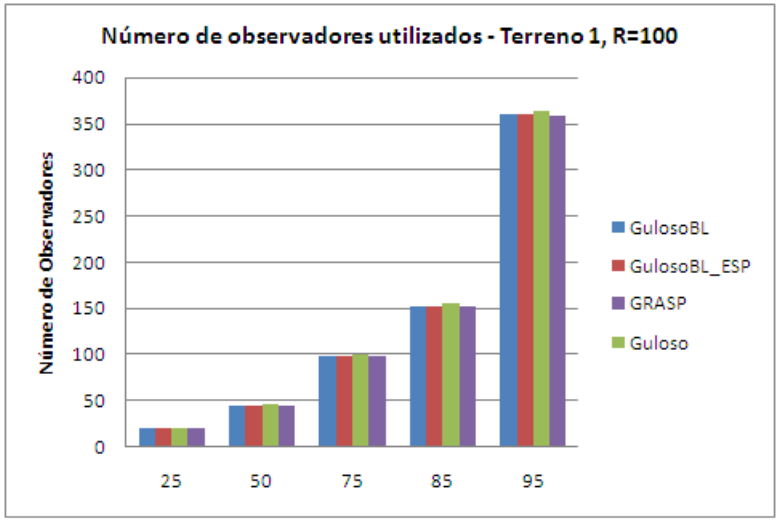
Novamente, assim como nos casos de teste do Terreno 1, todas as heurísticas propostas apresentaram soluções utilizando menos observadores do que o método

Terreno	R	VIX	*Guloso*	*GulosoBL*	*GulosoBL_ESP*	*GRASP*
1	100	25%	20	20 (0%)	20 (0%)	20 (0%)
		50%	46	45 (2%)	45 (2%)	45 (2%)
		75%	100	98 (2%)	98 (2%)	98 (2%)
		85%	155	153 (1%)	153 (1%)	152 (1%)
		95%	363	360 (1%)	360 (1%)	359 (1%)
1	200	25%	7	7 (0%)	7 (0%)	7 (0%)
		50%	18	18 (0%)	18 (0%)	18 (0%)
		75%	46	45 (2%)	45 (2%)	45 (2%)
		85%	77	76 (1%)	76 (1%)	76 (1%)
		95%	214	212 (1%)	212 (1%)	212 (1%)
1	300	25%	4	4 (0%)	4 (0%)	4 (0%)
		50%	11	11 (0%)	11 (0%)	11 (0%)
		75%	30	29 (3%)	29 (3%)	29 (3%)
		85%	54	52 (4%)	52 (4%)	52 (4%)
		95%	167	166 (1%)	166 (1%)	166 (1%)
2	100	25%	104	104 (0%)	104 (0%)	104 (0%)
		50%	207	207 (0%)	207 (0%)	207 (0%)
		75%	336	320 (5%)	321 (4%)	320 (5%)
		85%	413	384 (7%)	385 (7%)	382 (8%)
		95%	544	486 (11%)	487 (10%)	482 (11%)
2	200	25%	26	26 (0%)	26 (0%)	26 (0%)
		50%	52	52 (0%)	52 (0%)	52 (0%)
		75%	84	79 (6%)	79 (6%)	79 (6%)
		85%	104	93 (11%)	93 (11%)	93 (11%)
		95%	137	115 (16%)	114 (17%)	114 (17%)
2	300	25%	12	12 (0%)	12 (0%)	12 (0%)
		50%	23	23 (0%)	23 (0%)	23 (0%)
		75%	37	35 (5%)	35 (5%)	35 (5%)
		85%	46	41 (11%)	41 (11%)	41 (11%)
		95%	61	52 (15%)	51 (16%)	52 (15%)

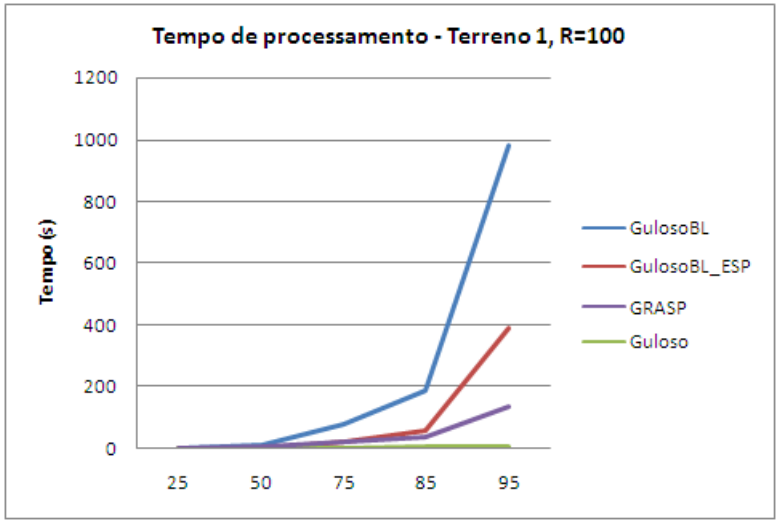
Tabela 4.1. Comparação do número de observadores utilizados pelas heurísticas para atingir diferentes índices de visibilidade nos terrenos 1 e 2. A diferença percentual entre as heurísticas propostas e o método *Guloso* é indicada entre parêntesis.

Terreno	R	VIX	*Guloso*	*GulosoBL*	*GulosoBL_ESP*	*GRASP*
1	100	25%	0 (1)	2 (2)	1 (0)	1 (3)
		50%	0 (0)	13 (10)	3 (3)	5 (14)
		75%	1 (1)	79 (62)	21 (20)	22 (36)
		85%	2 (1)	190 (148)	56 (47)	38 (39)
		95%	4 (3)	980 (777)	390 (247)	137 (105)
1	200	25%	0 (0)	2 (1)	1 (0)	1 (1)
		50%	0 (0)	11 (5)	6 (4)	3 (4)
		75%	1 (0)	65 (31)	39 (19)	46 (24)
		85%	3 (1)	151 (71)	85 (40)	37 (24)
		95%	7 (3)	1052 (472)	604 (235)	318 (127)
1	300	25%	0 (0)	1 (0)	1 (1)	1 (1)
		50%	1 (1)	8 (3)	7 (2)	8 (4)
		75%	2 (1)	43 (16)	33 (14)	49 (20)
		85%	4 (1)	131 (49)	110 (40)	113 (42)
		95%	11 (4)	1313 (477)	1108 (348)	492 (154)
2	100	25%	1 (3)	262 (166)	30 (19)	6 (49)
		50%	4 (6)	1024 (656)	136 (84)	25 (102)
		75%	10 (10)	5774 (3669)	988 (426)	1689 (884)
		85%	14 (12)	9996 (6340)	1648 (689)	1471 (703)
		95%	20 (15)	17806 (11163)	3301 (1315)	3922 (1640)
2	200	25%	0 (2)	58 (23)	6 (5)	4 (14)
		50%	3 (2)	221 (90)	27 (17)	10 (30)
		75%	8 (5)	1895 (698)	259 (99)	213 (105)
		85%	10 (6)	3165 (1174)	424 (154)	559 (215)
		95%	16 (8)	5998 (2228)	715 (250)	3003 (987)
2	300	25%	1 (1)	30 (9)	5 (2)	5 (7)
		50%	3 (2)	101 (32)	20 (9)	10 (12)
		75%	7 (3)	898 (277)	192 (61)	359 (114)
		85%	10 (4)	2283 (691)	384 (116)	632 (189)
		95%	15 (6)	4221 (1262)	719 (213)	1164 (334)

Tabela 4.2. Comparação do tempo de execução, em segundos, gasto pelas diferentes heurísticas considerando processamento em *CPU* e *GPU* (exibido entre parêntesis).

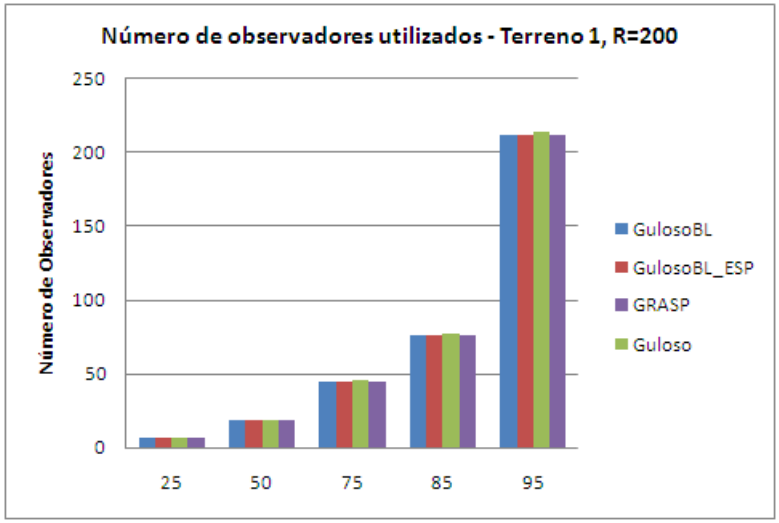


(a)

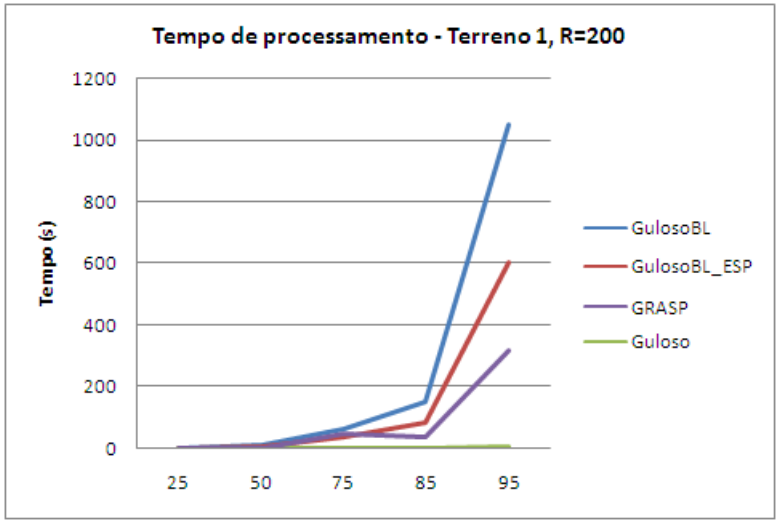


(b)

Figura 4.4. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 1, utilizando raio de interesse de 100 pontos.

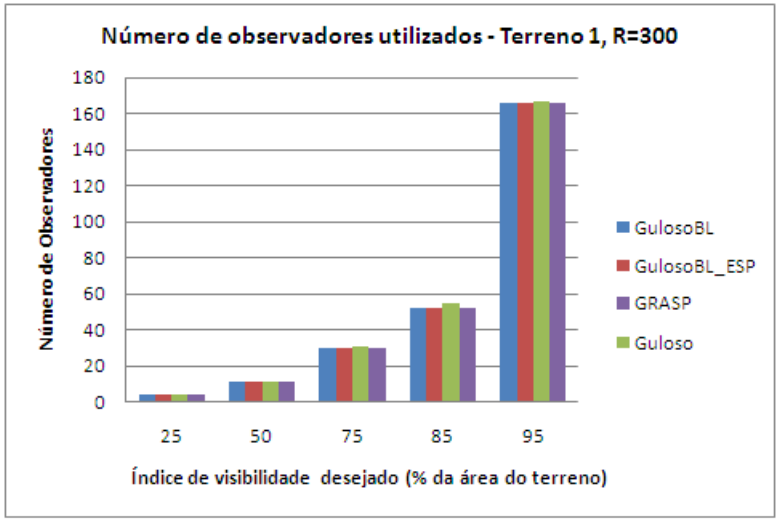


(a)

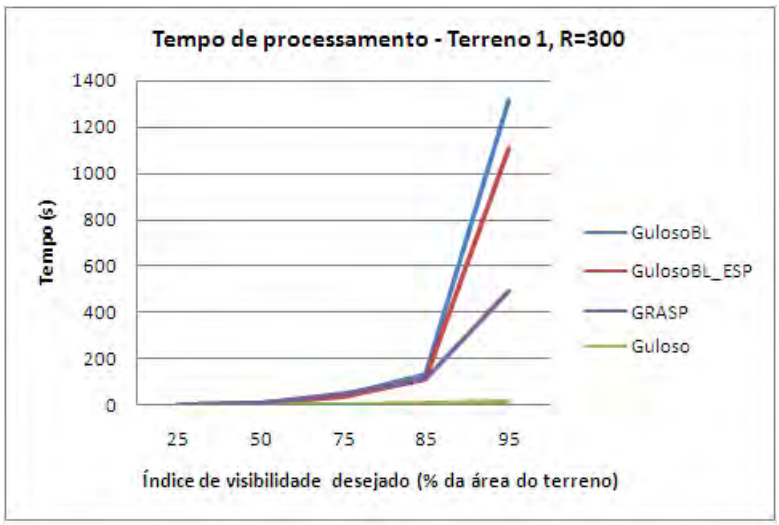


(b)

Figura 4.5. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 1, utilizando raios de interesse de 200 pontos.



(a)



(b)

Figura 4.6. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 1, utilizando raio de interesse de 300 pontos.

Guloso (exceto nos casos de teste onde o índice de visibilidade desejado é de 25% e 50%). O método *GulosoBL_ESP*, por exemplo, obteve soluções utilizando até $16,8\%$ menos observadores do que o métogo *Guloso* (isso pode ser observado no caso de teste que utiliza raio de 200 pontos e índice de visibilidade desejado igual a 95%).

Note que o crescimento acentuado do número de observadores que ocorreu no terreno 1 não se repetiu nesse terreno. Essa diferença se deve a características do relevo do Terreno 2, que possui áreas menos montanhosas.

Pelos gráficos também é possível observar que todas as heurísticas (inclusive o método *Guloso*) utilizaram a mesma quantidade de observadores em todos os casos de teste onde o índice de visibilidade desejado é pequeno (25% e 50%). Isso se deve à maior facilidade de se cobrir esse terreno posicionando observadores nas regiões com alto índice de visibilidade (é possível notar na Figura 4.3(b) que aproximadamente 50% da área do terreno é plana e, portanto, possui alto índice de visibilidade).

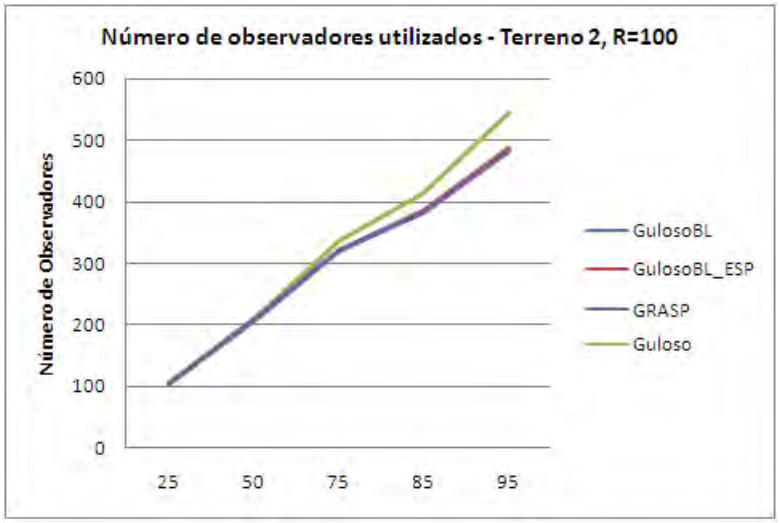
Nos casos de teste onde o índice de visibilidade desejado é maior (75% e 85%) todas as heurísticas propostas foram melhores do que o método *Guloso*. Note que, assim como no Terreno 1, a diferença na qualidade dos resultados do Terreno 2 variou pouco entre essas heurísticas.

Em relação ao tempo de processamento, o método *Guloso* apresentou, novamente, os menores tempos de execução e os métodos *GulosoBL_ESP* e *GRASP* foram os mais rápidos entre os métodos propostos.

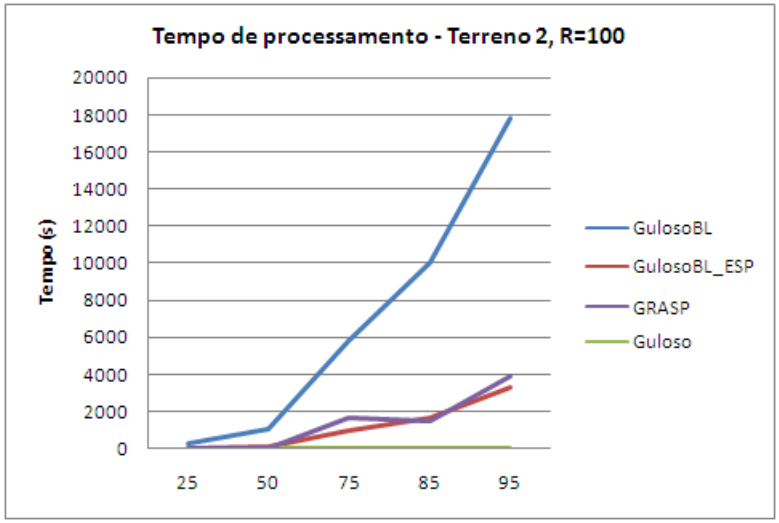
Os gráficos das Figuras 4.10 e 4.11 exibem uma comparação entre os tempos de processamento em *CPU* e *GPU* dos métodos *GulosoBL*, *GulosoBL_ESP*, *Guloso* e *GRASP* considerando o Terreno 2 e raio de interesse igual a 300 pontos.

Note que nesses casos de teste, à medida em que o índice de visibilidade desejado aumenta (eixo das abscissas), o processamento na *GPU* apresenta maiores ganhos de desempenho em relação à *CPU*, sendo que o processamento na placa gráfica chega a ser até $3,5$ vezes mais rápido do que o processamento na *CPU* (isso pode ser observado no caso da Figura 4.11(b), considerando o índice de visibilidade desejado de 95%). Esse ganho de desempenho ocorre porque quando o índice de visibilidade desejado é grande há a necessidade de posicionar uma quantidade maior de pontos e, portanto, as heurísticas processam mais *viewsheds*. Com isso, as operações que foram otimizadas para a *GPU* são utilizadas mais vezes.

Os gráficos das Figuras 4.12 e 4.13 exibem uma comparação semelhante à das Figuras 4.10 e 4.11 mas dessa vez é utilizado um raio de interesse igual a 100 pontos. Note que, nesse caso, o ganho de desempenho da *GPU* é menor do que no caso de teste que considera um raio de interesse de 300 pontos. Isso ocorre porque, com o raio de interesse menor, as estruturas de dados dos *viewsheds* possuem menos elementos

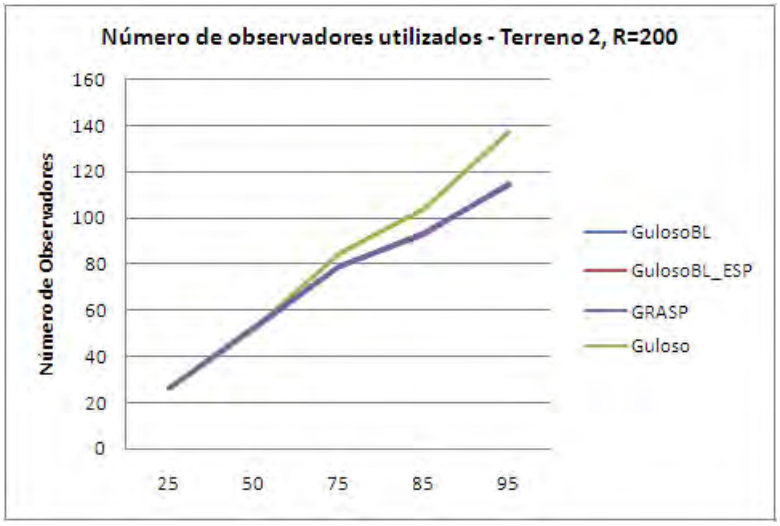


(a)

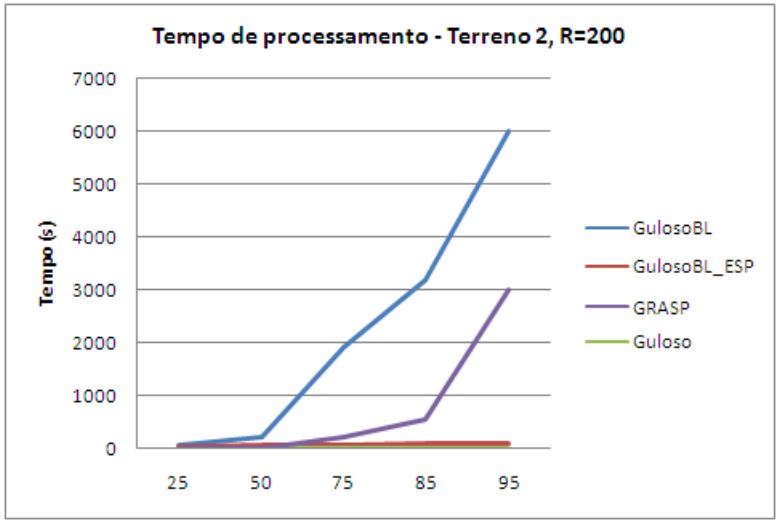


(b)

Figura 4.7. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 2, utilizando raio de interesse de 100 pontos.

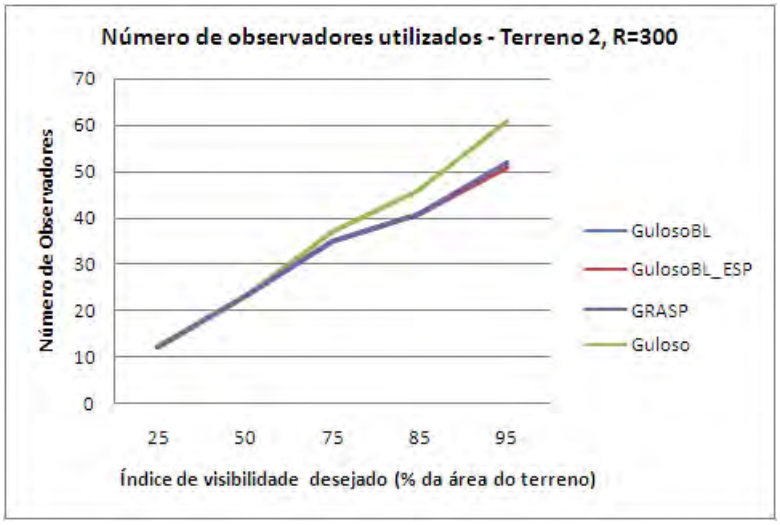


(a)

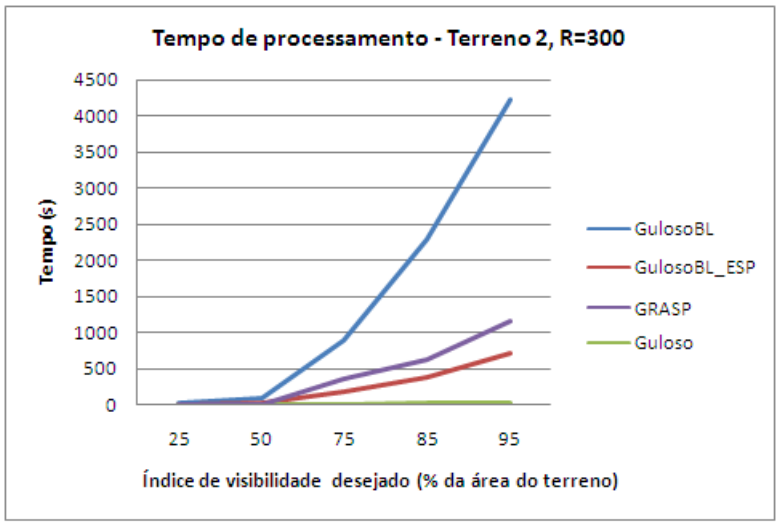


(b)

Figura 4.8. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 2, utilizando raios de interesse de 200 pontos.

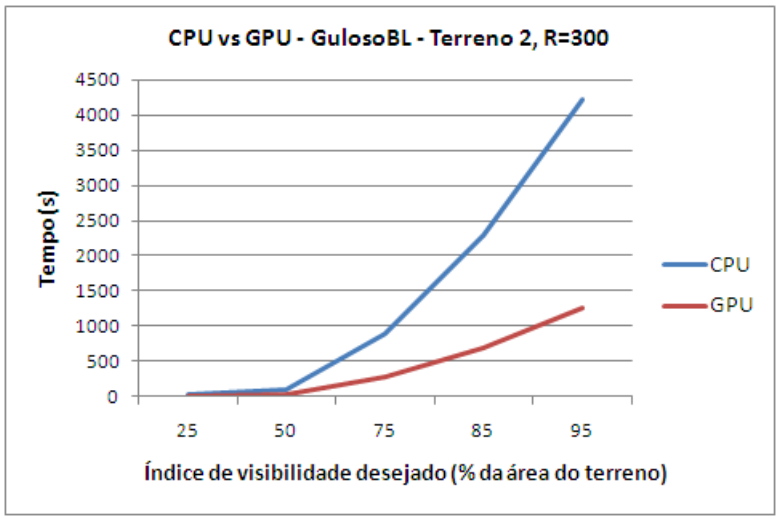


(a)

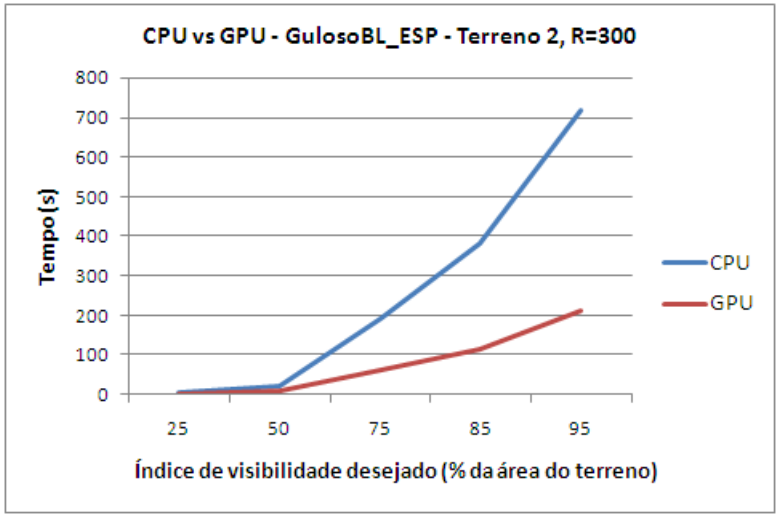


(b)

Figura 4.9. Número de observadores utilizados e tempo de processamento dos métodos *GulosoBL*, *GulosoBL_ESP*, *GRASP* e *Guloso* no Terreno 2, utilizando raio de interesse de 300 pontos.

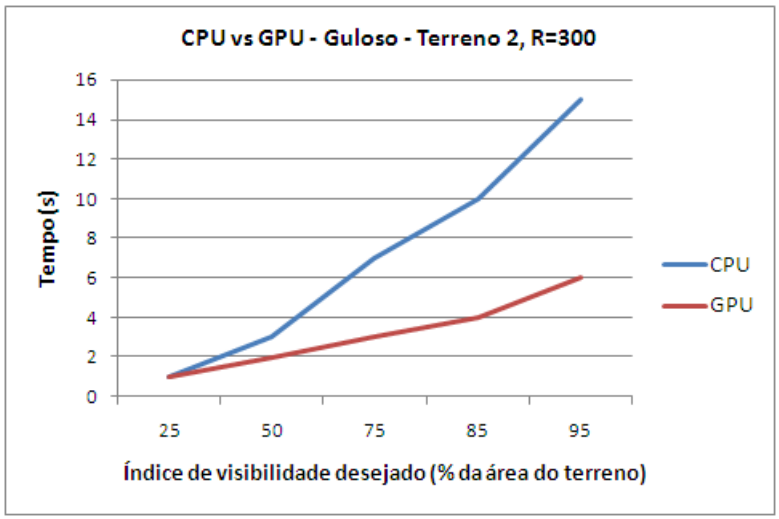


(a)

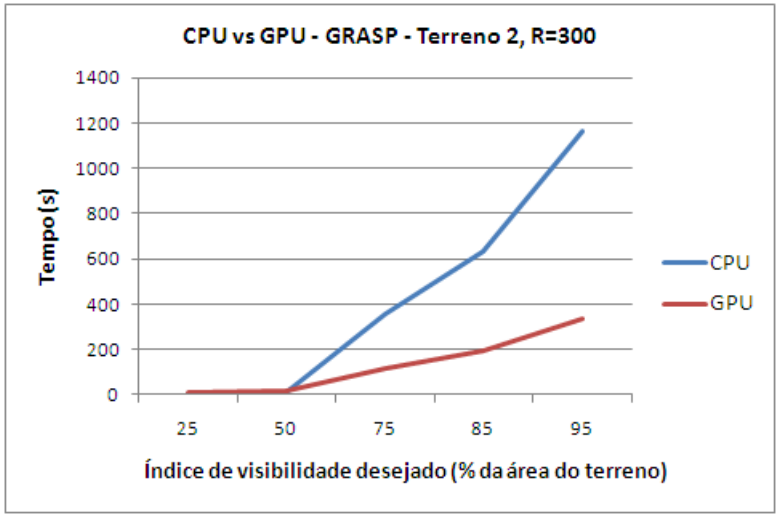


(b)

Figura 4.10. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *GulosoBL* (a) e *GulosoBL_ESP* (b) no Terreno 2, utilizando *raio de interesse* de 300 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.

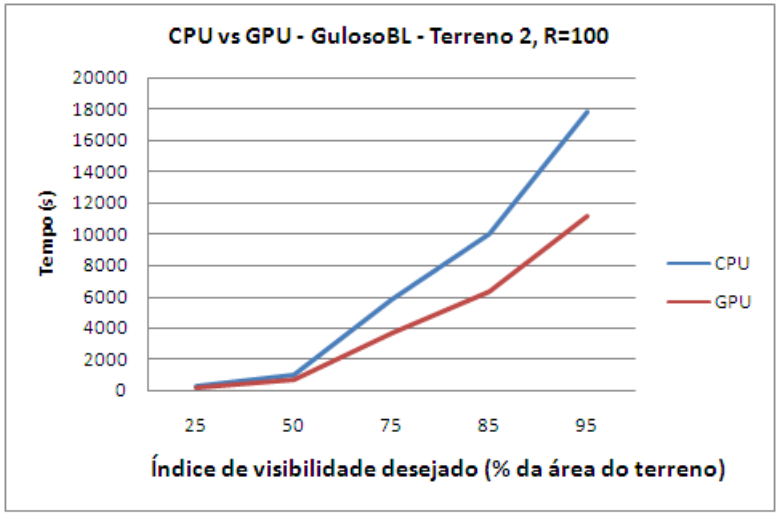


(a)

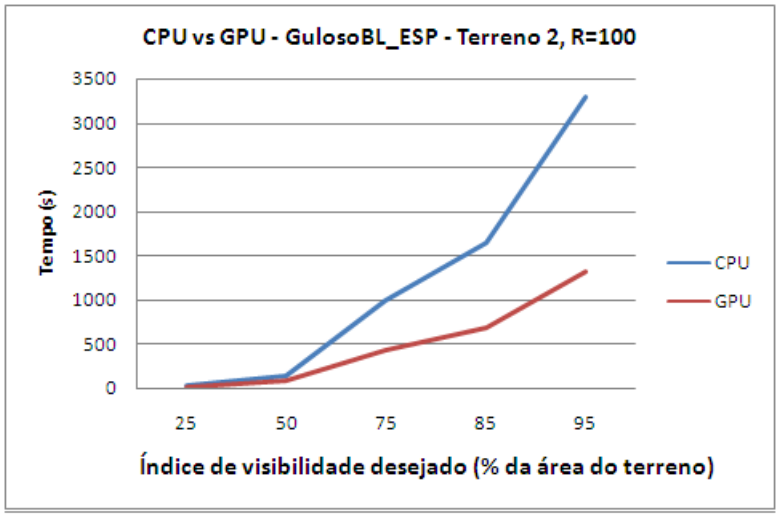


(b)

Figura 4.11. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *Guloso* (a) e *GRASP* (b) no Terreno 2, utilizando *raio de interesse* de 300 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.



(a)

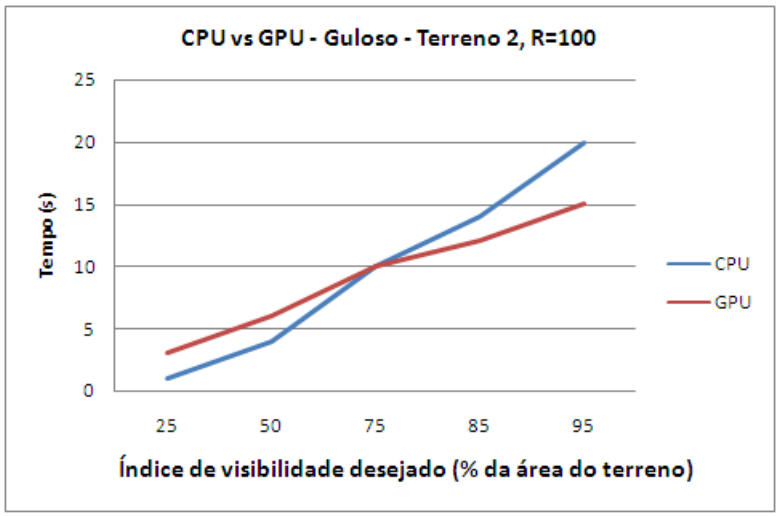


(b)

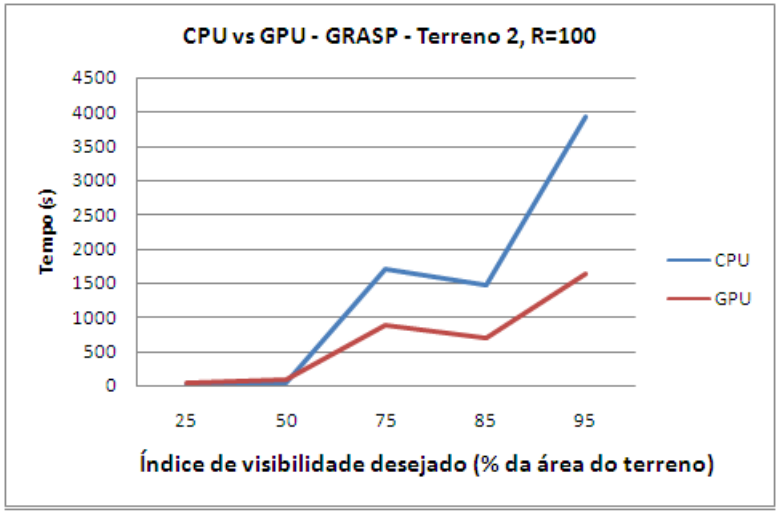
Figura 4.12. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *GulosoBL* (a) e *GulosoBL_ESP* (b) no Terreno 2, utilizando *raio de interesse* igual a 100 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.

e, portanto, a *GPU* realiza menos cálculos quando uma operação é aplicada nessas estruturas. Assim, ao realizar menos cálculos, a vantagem da *GPU* sobre a *CPU* é menor (como descrito na Seção 3.5.3).

As Figuras 4.14 e 4.15 exibem gráficos comparando o processamento na *CPU* com o processamento na *GPU* no Terreno 1 e utilizando raio de interesse de 100 pontos. Note que o ganho de desempenho da *GPU* é ainda menor do que o ganho obtido no caso de teste anterior. Esse ganho menor ocorre porque, além do raio de interesse ser pequeno (100 pontos), o terreno utilizado nos testes possui dimensões menores do que

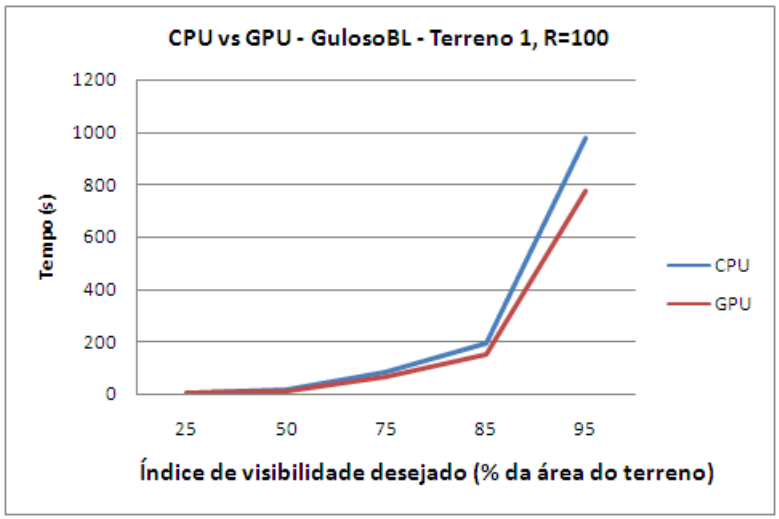


(a)

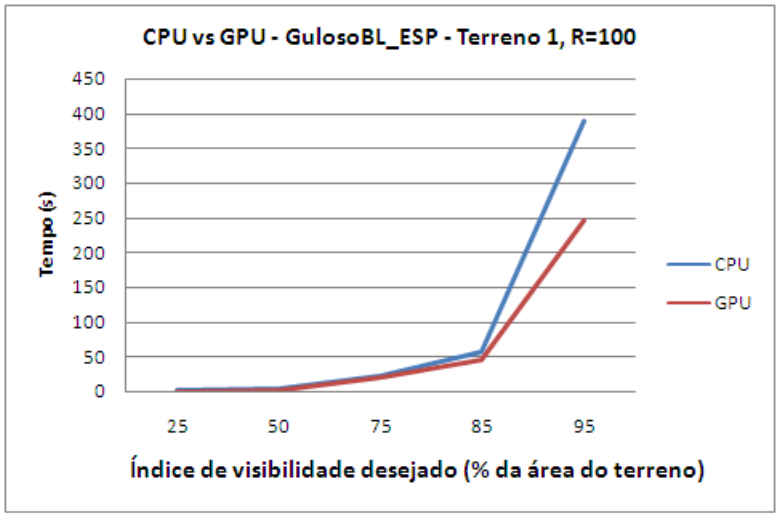


(b)

Figura 4.13. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *Guloso* (a) e *GRASP* (b) no Terreno 2, utilizando *raio de interesse* igual a 100 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.



(a)



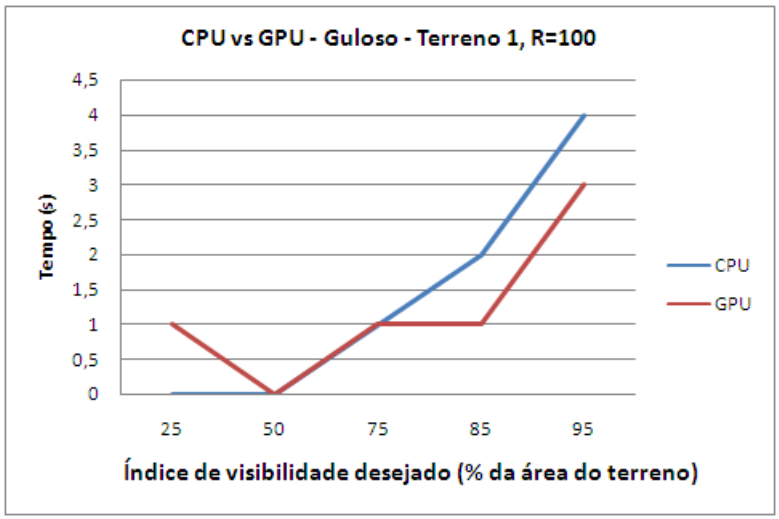
(b)

Figura 4.14. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *GulosoBL* (a) e *GulosoBL_ESP* (b) no Terreno 1, utilizando *raio de interesse* igual a 100 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.

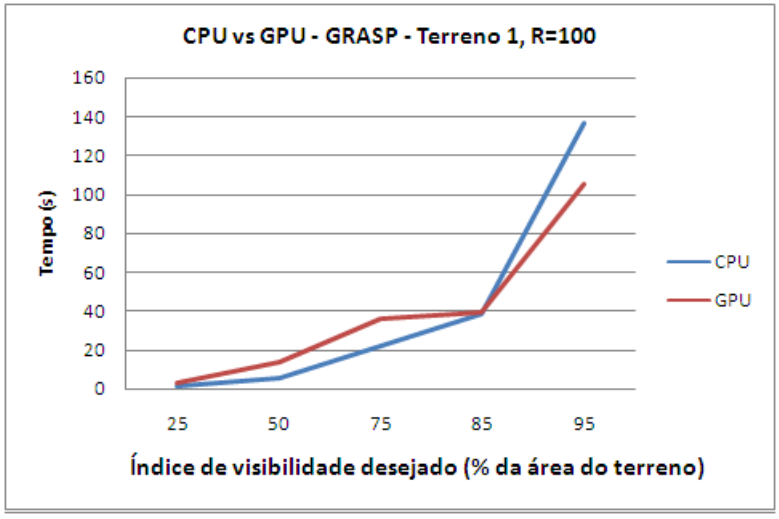
o Terreno 2 e, portanto, a aplicação de operações sobre o *viewshed* acumulado desse terreno envolve menos cálculos, diminuindo, com isso, a vantagem da *GPU*.

4.2 Experimentos considerando memória externa

Os testes realizados para avaliar o método de posicionamento de observadores em memória externa foram realizados em dois terrenos com dimensões 4804×4804 células obtidos a partir de dados do SRTM [30].



(a)



(b)

Figura 4.15. Comparação do tempo de processamento das versões para *CPU* e *GPU* dos métodos *Guloso* (a) e *GRASP* (b) no Terreno 1, utilizando *raio de interesse* igual a 100 pontos e considerando diferentes índices de visibilidade desejados. Note que os gráficos não estão na mesma escala.

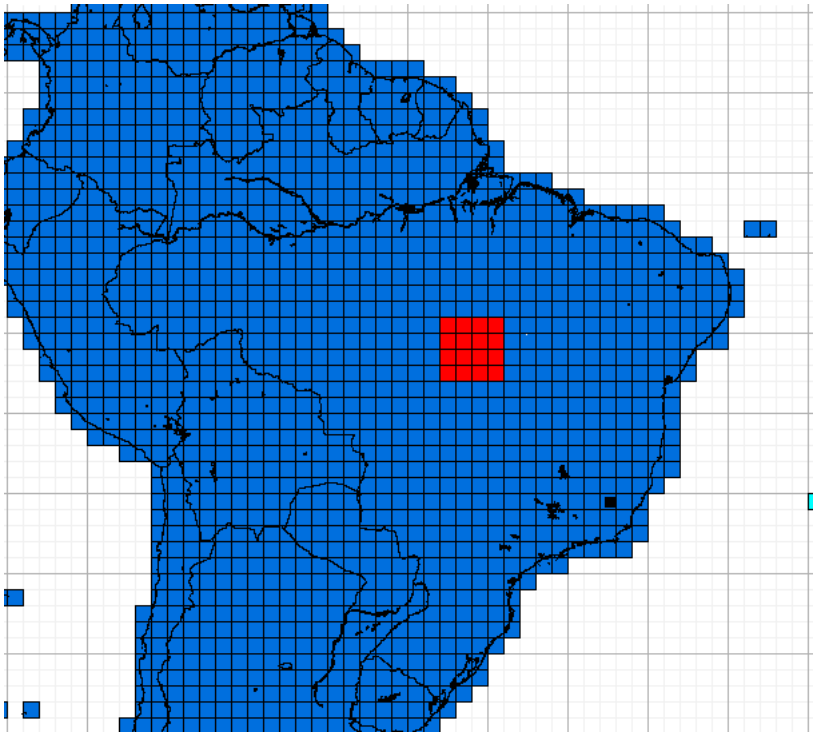


Figura 4.16. Mapa com destaque (em vermelho) para os blocos de dados referentes ao Terreno 4. Adaptado de [30]

Visto que a página do SRTM não disponibiliza dados de elevação do Brasil em blocos de dimensões maiores do que 1201 × 1201 células, um dos terrenos foi gerado utilizando o método de interpolação desenvolvido por Coelho e Andrade [7] para aumentar o número de pontos do Terreno 1 da Seção 4.1, enquanto o segundo terreno foi gerado unindo-se 16 blocos com dimensões 1201 × 1201 células:

- Terreno 3: Terreno com dimensões 4804 × 4804 células obtido a partir da interpolação do Terreno 1 da Seção 4.1. A Figura 4.1 destaca o bloco de dados correspondente a esse terreno e a Figura 4.3(a) exibe uma visualização desse bloco.

- Terreno 4: Terreno com dimensões 4804 × 4804 células obtido unindo-se 16 blocos de dados contendo, cada bloco, 1201 × 1201 células. Esse terreno representa a região do centro do Brasil destacada na Figura 4.16. A Figura 4.17 exibe uma visualização desse bloco gerada a partir dos dados de elevação.

O conjunto inicial de pontos candidatos a receberem observadores, formado por 16000 pontos, foi obtido utilizado o método de seleção por índice de visibilidade proposto por Franklin et al [13]. Assim como na Seção 4.1, o *viewshed* desses pontos foi

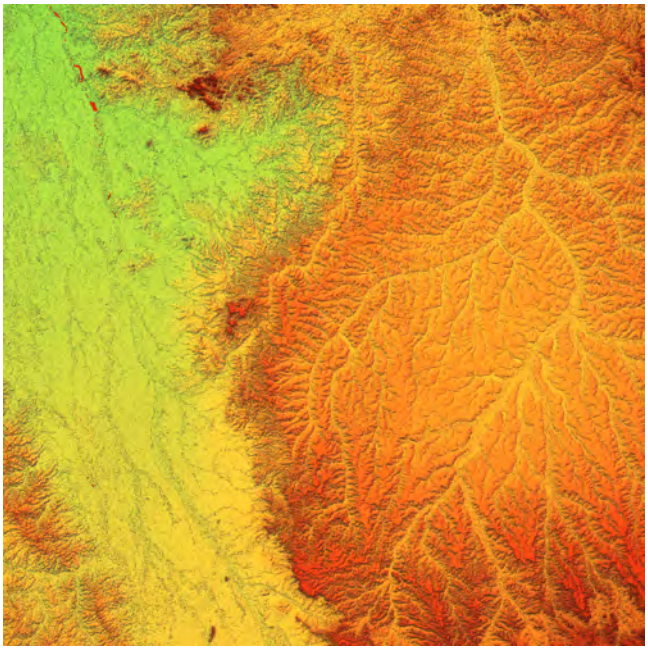


Figura 4.17. Visualização do Terreno 4 gerada utilizando o programa POV-Ray [29].

calculado utilizando o método *viewshed* [13] e a elevação acima do solo desses observadores foi definida como 30 metros.

É importante salientar que o processamento desse terreno pelo método *Site* original [13] exigiria mais de 42GB de memória, pois seria necessário armazenar todos os 16000 *viewsheds* dos pontos candidatos na memória e cada *viewshed* ocuparia $2,75$MB ($\frac{4804 \times 4804}{8}$ bytes).

Como o *EMSite* é o único método de posicionamento de observadores em memória externa de que temos conhecimento, ele foi comparado com a modificação trivial para memória externa do método *Site* que foi descrita na Seção 3.6.2.

A Tabela 4.3 compara o número de observadores utilizados pelos métodos *Site* adaptado e *EMSite* considerando raios de interesse de 100, 200 e 300 pontos (coluna R) e índices de visibilidade desejados de 25%, 50%, 75%, 85% e 95% (coluna VIX). A coluna *Melhora EMSite* indica a diferença percentual entre o número de observadores utilizados pelos dois métodos.

A Figura 4.18 exibe gráficos comparando o número de observadores necessários para se atingir determinados índices de visibilidade no Terreno 3 utilizando observadores com raios de interesse de 100, 200 e 300 pontos.

Nesses gráficos, é possível observar que a diferença entre os métodos *EMSite* e *Site* adaptado é maior nos casos de teste onde o *raio de interesse* é maior. Isso ocorre porque raios de interesse maiores fazem com que uma parte maior dos *viewsheds*

Terreno	R	VIX Desejado	*EMSite*	*Site*	Melhora *EMSite*
3	100	25%	205	209	1,9%
		50%	426	439	3,0%
		75%	732	767	4,6%
		85%	938	997	5,9%
		95%	1388	1483	6,4%
3	200	25%	60	66	9,1%
		50%	130	146	11,0%
		75%	242	266	9,0%
		85%	329	363	9,4%
		95%	536	606	11,6%
3	300	25%	30	39	23,1%
		50%	68	83	18,1%
		75%	134	166	19,3%
		85%	191	236	19,1%
		95%	342	418	18,2%
4	100	25%	186	192	3,1%
		50%	374	387	3,4%
		75%	600	639	6,1%
		85%	740	796	7,0%
		95%	1001	1083	7,6%
4	200	25%	47	51	7,8%
		50%	96	105	8,6%
		75%	158	176	10,3%
		85%	196	222	11,7%
		95%	275	323	14,9%
4	300	25%	22	39	43,6%
		50%	45	52	13,5%
		75%	76	90	15,6%
		85%	95	114	16,7%
		95%	145	185	21,6%

Tabela 4.3. Comparação do número de observadores utilizados pelo *Site* adaptado e pelo *EMSite* para atingir os índices de visibilidade desejados (coluna *VIX*) considerando observadores posicionados a $30m$ acima do solo.

de pontos próximos às bordas de cada região que está sendo processada fique fora dessa região. Como o método *Site* adaptado não calcula de forma correta a área de contribuição desses *viewsheds* (como afirmado na Seção 3.6), o método utiliza uma quantidade maior de observadores para atingir o índice de visibilidade desejado no terreno.

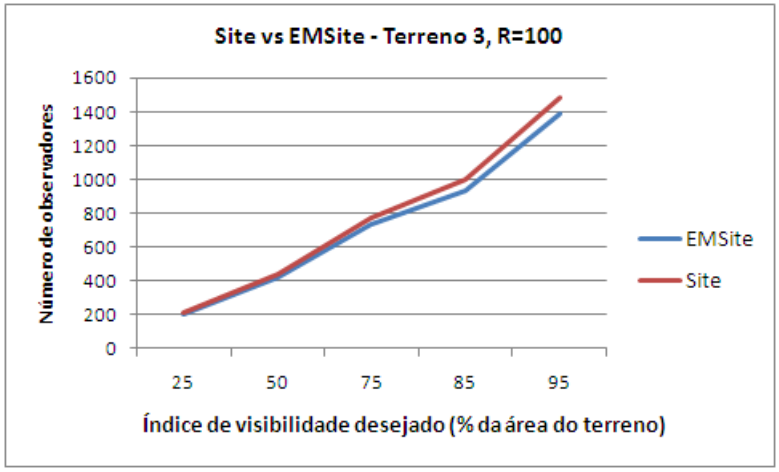
A Figura 4.19 exibe gráficos comparando o número de observadores necessários para se atingir determinados índices de visibilidade no Terreno 4 utilizando observadores com raios de interesse de 100 , 200 e 300 pontos. Observe que, assim como ocorre nos gráficos da Figura 4.18, a diferença entre o número de observadores utilizados pelos métodos cresce à medida em que o *raio de interesse* aumenta.

Note que, no caso de teste que considera raio de interesse igual a 300 pontos e índice de visibilidade desejado igual a 25%, a diferença entre os métodos foi de $43,6\%$ (algo bastante significativo). Isso ocorreu porque o método *Site* adaptado tentou atingir a cobertura de 25% em todos os subterrenos e, como foram posicionados poucos *viewsheds* e esses *viewsheds* possuíam muita área visível (devido ao tamanho do raio de interesse e a características do relevo do terreno), a cobertura atingida em alguns subterrenos passou de 25%. O *EMSite*, por outro lado, posicionou menos observadores em alguns subterrenos para compensar a cobertura extra atingida em outros subterrenos.

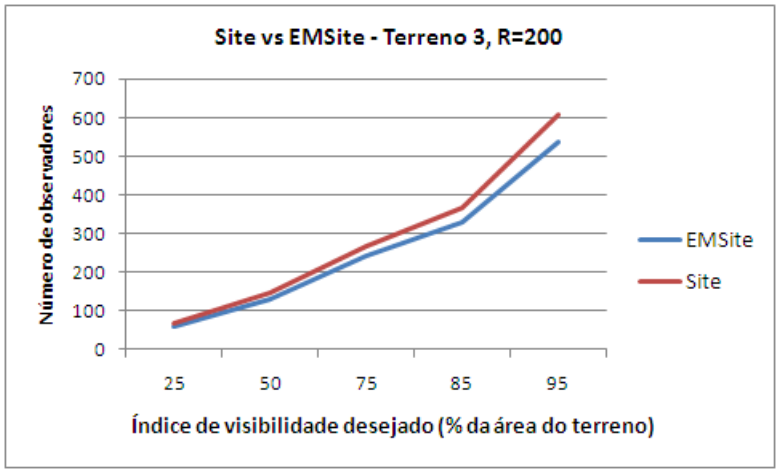
A Tabela 4.4 exibe o tempo de processamento (em segundos) dos métodos *Site* adaptado e *EMSite*. Devido à sua simplicidade, o método *Site* adaptado é mais rápido que o *EMSite* em quase todos os casos de teste. Para facilitar a visualização dessa diferença a coluna *Razão EMSite/Site* indica quantas vezes o método *Site* adaptado é mais rápido que o *EMSite*.

Os gráficos da Figura 4.20 apresentam o tempo de processamento utilizado pelos dois métodos considerando o Terreno 3 e diferentes raios de interesse. Observe que, à medida que o índice de visibilidade desejado aumenta, a diferença entre o tempo de processamento dos métodos também aumenta. Isso ocorre porque o método *Site* adaptado é aplicado apenas uma vez em cada subterreno. Por outro lado, quanto maior o índice de visibilidade desejado, mais vezes o método *EMSite* tende a processar um terreno.

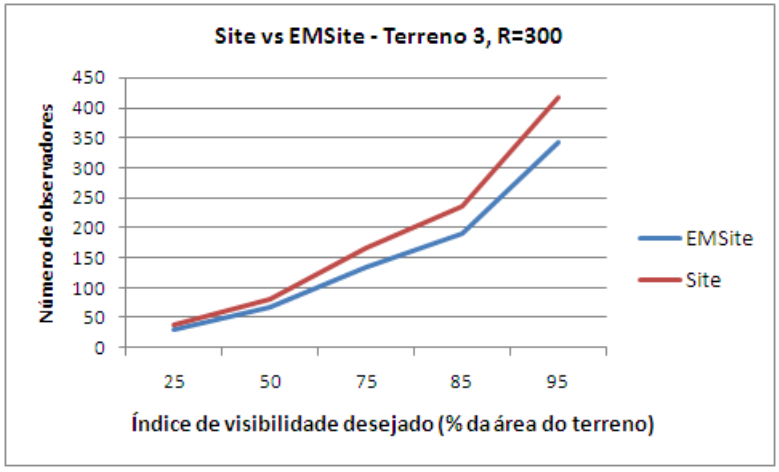
Outra tendência que pode ser observada nesses gráficos é que a diferença entre o tempo de processamento dos métodos é maior quando o *raio de interesse* cresce. Note que, no gráfico da Figura 4.20(c) o *EMSite* chega a ser $5,2$ vezes mais lento do que o método *Site* adaptado (porém, alcança uma economia de 18% no número de observadores utilizados). Essa diferença ocorre porque o método *EMSite* adiciona uma borda aos subterrenos que serão processados e a largura dessa borda é determinada pelo *raio de interesse*. Portanto, o *EMSite* processa subterrenos maiores nos casos de



(a)

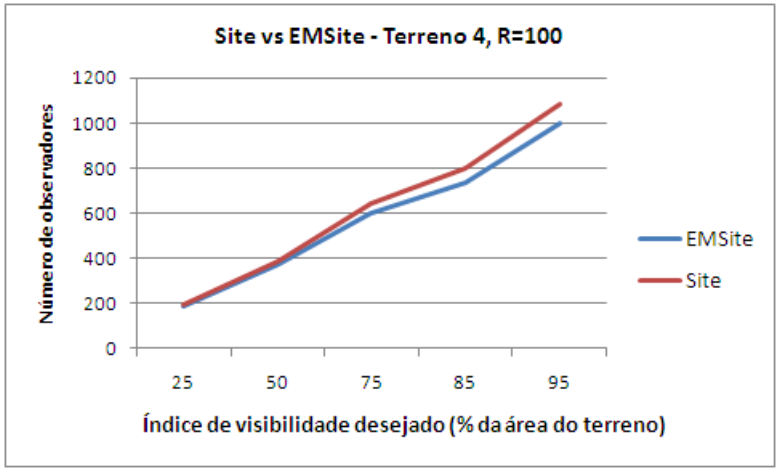


(b)

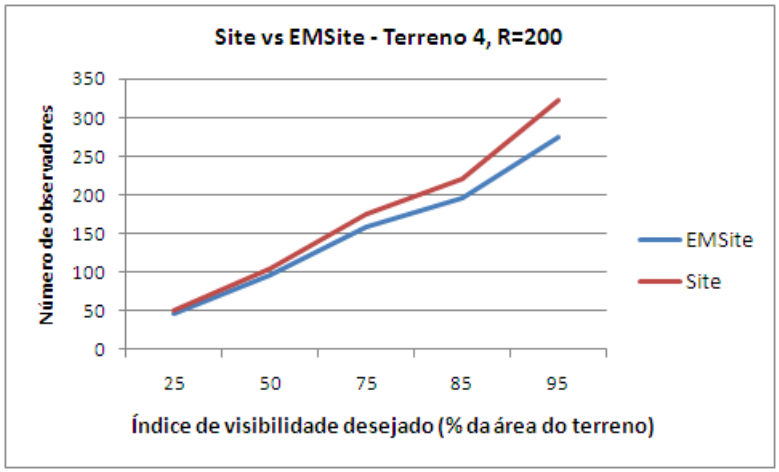


(c)

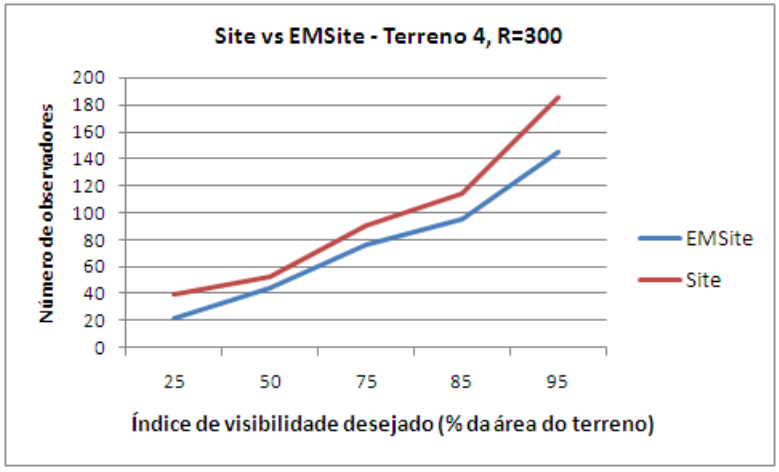
Figura 4.18. Comparação do número de observadores utilizados pelos métodos *EMSite* e *Site* adaptado para posicionar observadores no Terreno 3 utilizando diferentes índices de visibilidade desejados. Foram considerados raios de interesse de 100 (a), 200 (b) e 300 (c) pontos. Note que os gráficos não estão na mesma escala.



(a)



(b)



(c)

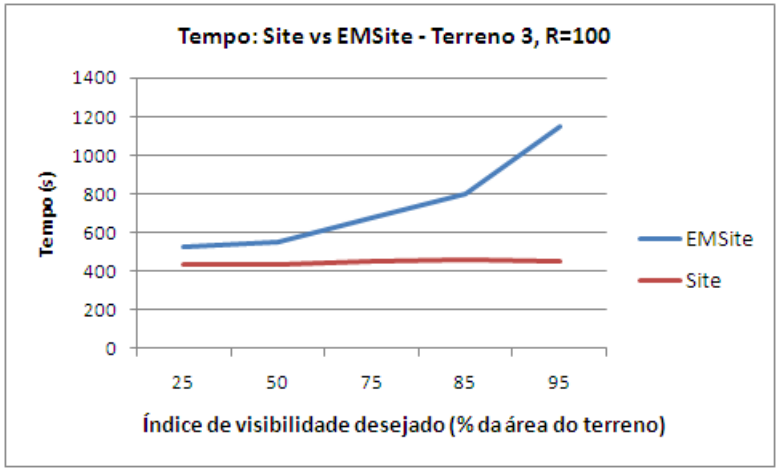
Figura 4.19. Comparação do número de observadores utilizados pelos métodos *EMSite* e *Site* adaptado para posicionar observadores no Terreno 4 utilizando diferentes índices de visibilidade desejados. Foram considerados raios de interesse de 100 (a), 200 (b) e 300 (c) pontos. Note que os gráficos não estão na mesma escala.

Terreno	R	VIX Desejado	*EMSite*	*Site*	*Razão EMSite/Site*
3	100	25%	525	434	1,2
		50%	553	432	1,3
		75%	677	446	1,5
		85%	802	456	1,8
		95%	1148	447	2,6
3	200	25%	1220	522	2,3
		50%	1306	531	2,5
		75%	1502	533	2,8
		85%	1697	530	3,2
		95%	2152	538	4
3	300	25%	2347	637	3,7
		50%	2437	622	3,9
		75%	2633	629	4,2
		85%	2811	628	4,5
		95%	3303	638	5,2
4	100	25%	498	523	1,0
		50%	510	509	1,0
		75%	597	516	1,2
		85%	694	517	1,3
		95%	952	521	1,8
4	200	25%	1158	711	1,6
		50%	1180	721	1,6
		75%	1262	745	1,7
		85%	1345	722	1,9
		95%	1536	729	2,1
4	300	25%	2309	936	2,5
		50%	2345	951	2,5
		75%	2431	944	2,6
		85%	2492	946	2,6
		95%	2647	951	2,8

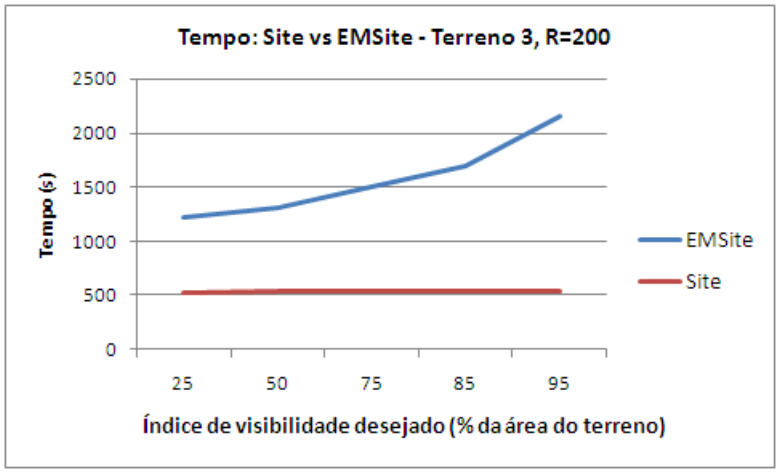
Tabela 4.4. Comparação do tempo de processamento, em segundos, dos métodos *Site* adaptado e pelo *EMSite* para atingir os índices de visibilidade desejados (coluna *VIX*) considerando observadores posicionados a $30m$ acima do solo. A coluna *Razão EMSite/Site* indica quantas vezes o *Site* adaptado foi mais rápido do que o *EMSite*.

teste que consideram raios grandes.

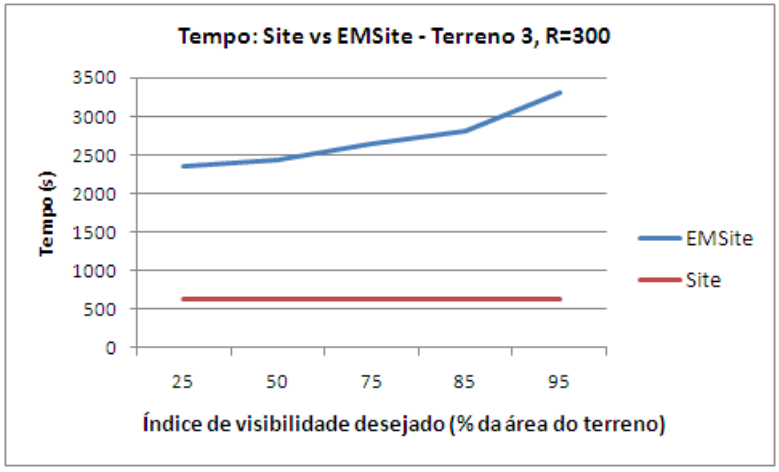
A Figura 4.21 exibe os tempos de processamento dos métodos *EMSite* e *Site* adaptado considerando o Terreno 4 e diferentes raios de interesse. Observe que as tendências observadas nos gráficos da Figura 4.20 se repetem nesses gráficos.



(a)

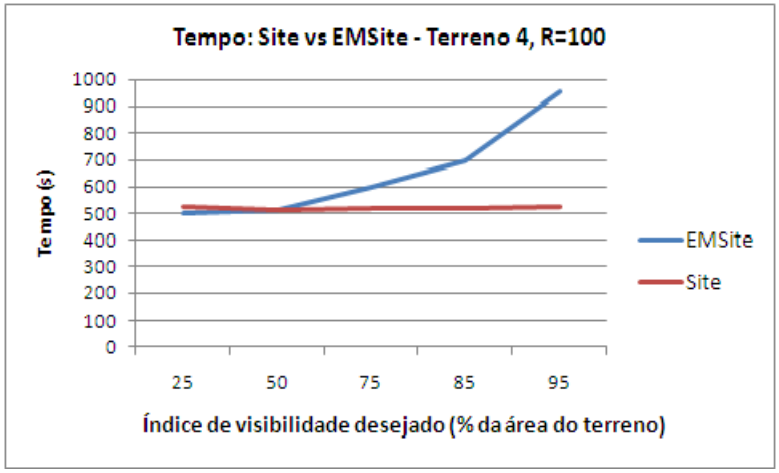


(b)

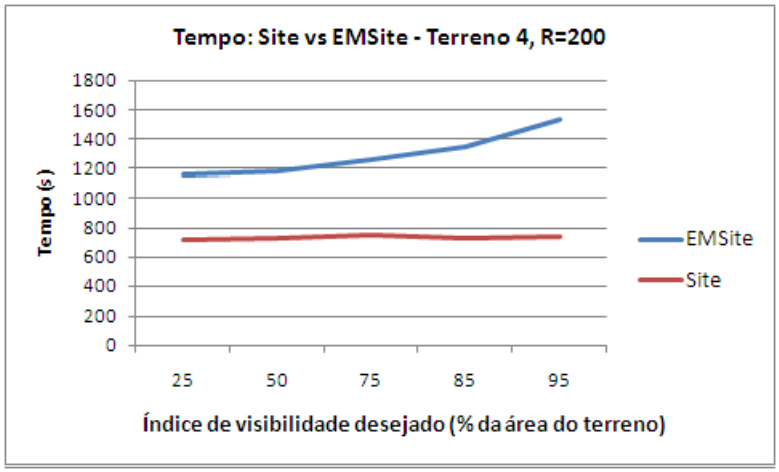


(c)

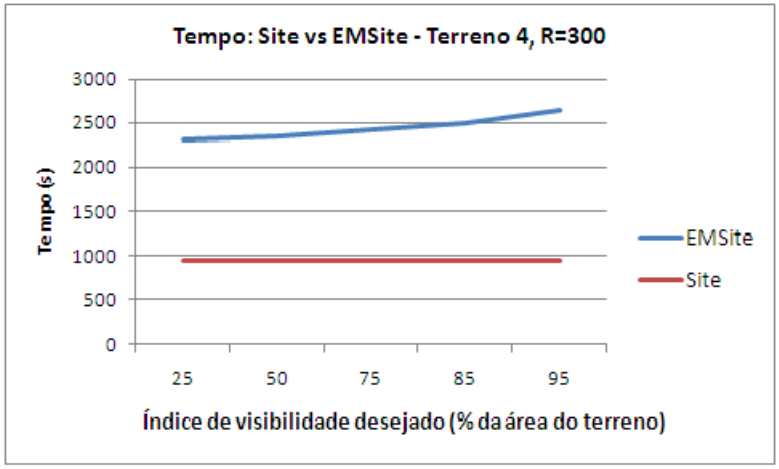
Figura 4.20. Comparação do tempo (em segundos) gasto pelos métodos *EMSite* e *Site* adaptado para posicionar observadores no Terreno 3 utilizando diferentes índices de visibilidade desejados. Foram considerados raios de interesse de 100 (a), 200 (b) e 300 (c) pontos. Note que os gráficos não estão na mesma escala.



(a)



(b)



(c)

Figura 4.21. Comparação do tempo (em segundos) gasto pelos métodos *EMSite* e *Site* adaptado para posicionar observadores no Terreno 4 utilizando diferentes índices de visibilidade desejados. Foram considerados raios de interesse de 100 (a), 200 (b) e 300 (c) pontos. Note que os gráficos não estão na mesma escala.

Capítulo 5

Conclusões

Neste trabalho, foram propostas diferentes abordagens para a solução aproximada do problema de posicionamento de observadores em terrenos representados por modelos digitais de elevação.

Foram desenvolvidas três heurísticas para o posicionamento de observadores em memória interna. Conforme observado nos testes realizados, os métodos propostos apresentam soluções até 17% melhores do que a heurística gulosa utilizada por outros autores [15; 13]. No entanto, é importante ressaltar que o tempo de processamento das heurísticas propostas é maior do que o tempo de execução da heurística gulosa. Com isso, aplicações onde um tempo de processamento maior é tolerável podem tirar proveito da economia proporcionada por essas soluções. Em problemas como, por exemplo, o planejamento da distribuição de torres de telefonia celular a economia de até mesmo um observador pode representar ganhos financeiros consideráveis.

O tempo de processamento das heurísticas implementadas foi melhorado utilizando-se técnicas para processar de forma paralela na *GPU* as estruturas de dados utilizadas pelas heurísticas, aproveitando, assim, o poder de processamento das placas gráficas atuais. Com isso, foi possível tornar as implementações desses métodos até 4 vezes mais rápidas do que as implementações correspondentes que utilizam apenas a *CPU*.

Por fim, foi desenvolvido um método capaz de realizar o posicionamento de observadores em instâncias do problema que não poderiam ser processadas na memória principal dos computadores. Esse método foi capaz de obter soluções que utilizam, em média, 12% menos observadores do que as soluções obtidas a partir de uma modificação direta para a memória externa do método de posicionamento de observadores utilizado por Franklin et al [15; 13]. Apesar do método proposto ser mais lento do que o outro método, assim como no caso do processamento em memória interna, a economia propiciada pelas soluções de melhor qualidade podem compensar essa diferença de tempo.

Os resultados descritos neste trabalho foram apresentados e publicados em im-

portanes congressos da área. Mais precisamente, as heurísticas de posicionamento de observadores em memória interna foram publicados no artigo [25], que foi apresentado no "10^o Simpósio Brasileiro de GeoInformática" (GEOINFO 2010). Além disso, a heurística para posicionamento de observadores em terrenos armazenados em memória externa foi publicada nos artigos [24] e [26] que foram apresentados, respectivamente, na "XXXV Conferência Latino Americana de Informática" (CLEI 2009) e na "10^{th} International Conference on Hybrid Intelligent Systems" (HIS 2010).

5.1 Trabalhos futuros

São apresentadas a seguir algumas sugestões de trabalhos futuros envolvendo o problema de posicionamento de observadores:

- Desenvolvimento de uma interface gráfica para facilitar o uso das heurísticas: Uma melhoria interessante para esse trabalho consiste em desenvolver uma interface gráfica onde os usuários possam visualizar as soluções geradas pelas diferentes heurísticas e, além disso, editar as soluções manualmente (por exemplo, adicionar um observador em um certo ponto, modificar a posição de um observador, remover um observador, entre outras funcionalidades).

- Implementação das heurísticas de posicionamento de observadores para processamento em *Clusters* de computadores: um *Cluster* é um grupo de computadores que são conectados através de uma rede e executam tarefas em paralelo Uma possível forma de se obter maiores ganhos de desempenho consiste em implementar as heurísticas de posicionamento de observadores de modo que elas possam ser executadas de forma paralela em *Clusters*. Por exemplo, o método *GRASP* poderia ser paralelizado para que cada nodo do *Cluster* possa executar uma iteração do método de forma paralela com os outros nodos.

- Adaptação das heurísticas para processamento em *FPGAs*: Outra possível forma de melhorar o desempenho das heurísticas propostas nesse trabalho consiste em adaptá-las para processamento em *FPGAs*, que são processadores cujos circuitos podem ser reconfigurados pelo usuário para que eles possam ser dedicados a resolverem tarefas específicas. Assim, os circuitos de uma *FPGA* podem ser programados para formar um processador especializado em realizar operações sobre os *viewsheds* (por exemplo, união, cálculo de área de contribuição, etc) de forma eficiente.

- Desenvolvimento de um método de posicionamento de observadores que otimiza outras características dos observadores: Outra melhoria para este trabalho consiste em adaptar os métodos de posicionamento de observadores para otimizar diferentes características destes observadores (elevação, raio de interesse, etc). Por exemplo, no caso do posicionamento de torres de telefonia celular o método poderia associar um custo à elevação dessas torres acima do terreno e, assim, o posicionamento seria realizado de modo a minimizar o número de torres utilizadas e também a dimensionar essas torres de forma otimizada.

Referências Bibliográficas

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] R. M. Aiex, S. Binato, and M. G. C. Resende. Parallel grasp with path-relinking for job shop scheduling. *Parallel Comput.*, 29:393–430, April 2003.

[3] Marcus V. A. Andrade, Salles V. G. Magalhães, Mirella A. Magalhães, W. Randolph Franklin, and Barbara M. Cutler. Efficient viewshed computation on terrain in external memory. *GeoInformatica*, 2009. "`http://www.springerlink.com/content/p1783648185g1252/` (Acessado em Novembro 2010)".

[4] R. G. I. Arakaki. *Heurística de Localização-alocação para problemas de localização de facilidades*. Tese de doutorado em computação aplicada, Instituto Nacional de Pesquisas Espaciais, INPE, 2002.

[5] Paul E. Black. Set cover. In Paul E. Black, editor, *Dictionary of Algorithms and Data Structures [online]*. U.S. National Institute of Standards and Technology, 17 Dec 2004. `http://www.itl.nist.gov/div897/sqg/dads/HTML/setcover.html`, [Acessado em Novembro 2010].

[6] Patrice Calégari, Frédéric Guidec, Pierre Kuonen, and Frank Nielsen. Combinatorial optimization algorithms for radio network planning. *Theor. Comput. Sci.*, 263(1-2):235–265, 2001.

[7] Jéferson R. P. Coêlho and Marcus V. A. Andrade. Modelagem de terrenos usando splines. Relatório de trabalho de conclusão de curso, Universidade Federal de Viçosa, UFV, 2010.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[9] Lawrence Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[10] Jason Deane, Terry Rakes, and Loren Rees. Efficient heuristics for wireless network tower placement. *Information Technology and Management*, 10:55–65, 2009. 10.1007/s10799-008-0046-x.

[11] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[12] C. A. Felgueiras. Modelagem numérica de terreno. In A. M. V. Monteiro In G. Câmara, C. Davis, editor, *Introdução à Ciência da Geoinformação*, volume 1. INPE, 2001.

[13] W. R. Franklin. Siting observers on terrain. In Springer-Verlag, editor, *In D. Richardson and P. van Oosterom editors, Advances in Spatial Data Handling: 10th International Symposium on Spatial Data Handling*, pages 109–120, 2002.

[14] W. R. Franklin and C. Ray. Higher isn't necessarily better - visibility algorithms and experiments. In *6th Symposium on Spatial Data Handling*, pages 751–770, Edinburgh, Scotland, 1994. Taylor & Francis.

[15] W. R. Franklin and C. Vogt. Tradeoffs when multiple observer siting on large terrain cells. In Springer-Verlag, editor, *12th International Symposium on Spatial Data Handling*, pages 845–861, 2006.

[16] Micheal F. Goodchild and Jay Lee. Coverage problems and visibility regions on topographic surfaces. *Ann. Oper. Res.*, 18(1-4):175–186, 1989.

[17] Pawan Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, chapter 21, pages 197–208. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007.

[18] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[19] H. Haverkort, L. Toma, and Y. Zhuang. Computing visibility on terrains in external memory. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments / Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALCO)*, 2007.

[20] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[21] Young hoon Kim, Sanjay Rana, and Steve Wise. Exploring multiple viewshed analysis using terrain features and optimisation techniques. *Computers & Geosciences*, 30:1019–1032, 2004.

[22] R. Laurini and D. Thompson. *Fundamentals od Spatial Information Systems*. Academic Press, 1992.

[23] X. Li. A hybrid algorithm for terrain simplification. Master thesis, Department of Computer Science - The University of British Columbia, 2003.

[24] Salles V. G. Magalhães and Marcus V. A. Andrade. Heurística para o posicionamento de observadores em terrenos armazenados em memória externa. In *Proc. 35th Latin American Informatics Conference (CLEI)*, 2009.

[25] Salles V. G. Magalhães, Marcus V. A. Andrade, and Chaulio Ferreira. Heuristics to site observers in a terrain represented by a digital elevation matrix. In *Anais do XI Simposio Brasileiro de Geoinformatica*, 2010.

[26] Salles V. G. Magalhães, Marcus V. A. Andrade, Chaulio Ferreira, and W. R. Franklin. An optimization heuristic for siting observers in huge terrains stored in external memory. In *Proc. 10th International Conference on Hybrid Intelligent Systems*, Atlanta, EUA, 2010.

[27] CUDA Programming Guide. `http://www.nvidia.com/object/cuda_develop.html` (Acessado em Novembro 2010), 2007.

[28] CUDA Zone. `http://www.nvidia.com/object/cuda_home.html` (Acessado em Novembro 2010), 2007.

[29] Persistence of Vision (TM) Raytracer. `http://www.povray.org/download/` (Acessado em Novembro 2010), 2009.

[30] The Shuttle Radar Topography Mission (SRTM). `http://www2.jpl.nasa.gov/srtm/` (Acessado em Novembro 2010), 2007.

[31] M Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[32] G. Nagy. Terrain visibility. *Computers and Graphics*, 18(6):763–773, 1994.

[33] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[34] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*, 2008.

[35] Clark K. Ray. *Representing Visibility for Siting Problems*. PhD thesis, Rensselaer Polytechnic Institute, 1994.

[36] Mauricio G. C. Resende. Greedy randomized adaptive search procedures (grasp). *Journal of Global Optimization*, 6:109–133, 1999.

[37] Mauricio G. C. Resende and Renato F. Werneck. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10(1):59–88, 2004.

[38] SGI. *Standard Template Library Programmer's Guide*, 2005.

[39] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68:1381–1388, October 2008.

[40] Mauricio Solar, Víctor Parada, and Rodrigo Urrutia. A parallel genetic algorithm to solve the set-covering problem. *Comput. Oper. Res.*, 29:1221–1235, August 2002.

[41] L. Toma, L. Arge, and J. S. Vitter. I/o-efficient algorithms for problems on grid-based terrains. *J. Exp. Algorithmics*, 6:1, 2001.

[42] M. van Kreveld. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. In *Symposium on Spatial Data Handling*, pages 15–27, 1996.

[43] C. Vogt W. R. Franklin. Efficient multiple observer siting on large terrain cells. In *Third International Conference on Geographic Information Science*, University of Maryland, 2004.