TOWARDS OPTIMAL LINE OF SIGHT COVERAGE

by

Peter S. Gu

A dissertation submitted to Johns Hopkins University in conformity with the requirements for the degree of Doctor of Engineering

Baltimore, Maryland

August, 2022

Abstract

Maintaining the line of sight to a moving object or person over long distances is critical in many applications, e.g., mobile communications, security, surveillance. Determining the best places to position (or build) technologies is difficult because even small changes in the location can greatly affect the so-called *viewshed*, which is the collection of land areas within line of sight of a given observer. The need for multiple sensors or towers further complicates this problem, as they often need to work cooperatively to achieve the best possible coverage. This study proposes a novel approach that consists of three separate inventions: 1) An algorithm for calculating viewsheds from many sensors in parallel, 2) Introduction of a meaningful measure of quality for coverage to compare competing configurations; and 3) Optimization of that well-defined objective function to find the best suitable sensor parameters for practical applications. Preliminary results suggest unprecedented performance on a wide range of real terrains.

Primary Reader and Advisor: Tamás Budavári, PhD

Additional Readers: Randal Burns, PhD and Amanda Galante, PhD

Contents

List of Figures

Chapter 1

INTRODUCTION

Protecting large pieces of infrastructure, such as commercial airfields, oil fields, power plants, or commercial/industrial campuses over long distances in remote areas can be a challenge because of the vast areas of land where intrusions could occur and the relatively low chance that incidents could occur at any one area at any time [2] [3] [4]. It is likely prohibitively expensive to monitor all areas around a piece of infrastructure at all times. Therefore, sensor assets such as fixed towers with electro-optical (EO), infrared (IR), radar technologies, or flying assets such as unmanned aerial vehicles (UAVs) should be chosen and positioned carefully to have the greatest effect with minimal cost. Security guards can work together with the sensor network to cover small gaps with their patrol routes. The challenge then becomes how to determine the best location to place sensors in order to minimize gaps in their line of sight (LOS) coverage of the terrain.

There exists an equivalent problem where communications towers that rely on LOS are to be placed in a way that creates a path with as close to consistent coverage as possible when traveling through a remote area. We will adopt the terminology of the former, security application throughout most of this study but note that the methodology and results are analogous when applied to the communication problem.

Digital Elevation Models (DEMs) store the geographical data for viewshed calculations. DEM can contain high resolution gridded elevation data over wide areas. In particular, a Digital Surface Model (DSM) from LiDAR collections captures the elevation of the highest objects in each grid cell, which includes features like vegetation and buildings. As technology improves, the resolution of DSM data increases, even over large land areas, which necessitates the development of algorithms to process these data more efficiently in addition to simply increasing computing power [1].

Considering an object with a constant height above the ground, the set of geographic locations where that object can be placed and have LOS to an observer is called the observer's *viewshed* [1]. LOS can be blocked by terrain features, and is sometimes limited to a certain range R specified for the observer. It is also possible to calculate the observer's probability of detecting an object with respect to range given that it is within LOS, which creates a *probability of detection viewshed*.

Many efficient algorithms and software packages exist for calculating viewsheds and are used in a wide variety of applications. A common algorithm is called `r3` due to its run time of $O(R^3)$ [1]. This study uses the `r2` algorithm, which calculates an approximate viewshed in $O(R^2)$ time [1]. Optimization requires the calculation of large numbers of viewsheds, so each individual viewshed does not need to be as accurate as possible. Furthermore, the `r2` algorithm retains high accuracy close to observer but is less accurate farther from the observer, which is acceptable because at high ranges, observers have diminishing performance and are harder to model due to accumulation of numerical errors from the DSM and physical effects such as atmospheric turbulence. Both algorithms have efficient parallel processing implementations suited for computation on GPUs [1].

Next, consider the combination of multiple viewsheds for a set of observers. One can calculate the probability of an object being detected by any observer to create an aggregate probability of detection viewshed. Algorithms exist to calculate the total viewshed, the set of points visible to one or more observers, more efficiently than calculating each viewshed in series [5], but this only takes into account LOS. In order to calculate the probability of detection, we need to keep track of which observers each area has LOS to.

Depending on the terrain being investigated, optimizing LOS coverage based on a sensor's location on a map can be highly non-convex due to situations where moving an observer by a small amount drastically changes a viewshed, such as when an LOS obstruction is nearby, even as small as a tree, or when the observer is near the edge of a cliff, where a small movement could determine whether the whole cliff face is visible or not. This is a hard optimization problem where deterministic and convex optimizations have not provided satisfactory real-life solutions, so a previous study used simulated annealing (SA), limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS), and evolutionary algorithms to optimize an objective function consisting of the total probability of detection over the entire area of interest [6].

This study uses a larger terrain dataset and has a larger set of possible tower placements compared to previous studies such as [6], [7], and [8]. We place fewer towers, but each has longer range and the terrain has a higher resolution. Higher resolution terrain means that numerous smaller LOS obstructions such as trees can be resolved, which can cause viewsheds to change suddenly when moving observers by a short distance, making optimization methods that use gradients such as L-BFGS less reliable.

The objective function is also different. In [6], the objective is simply the sum over all cells of the aggregate probability of detection viewshed. This study takes this a step further, where we minimize coverage gap in order to provide more targeted coverage in applications of long range security and communications pathways. The proposed objective function defines a line of start cells and a line of end cells, then measures how well the aggregate probability of detection viewshed deters "movement" between the lines. This means that while the observers' locations are not optimized for maximum coverage, their coverage areas should have orientations and overlapping patterns that better fit realistic requirements. It does this using a maximum flow minimum cut formulation. The objective requires only a few shortest path calculations, which can theoretically be done in linear time on a planar graph like a DSM [9].

In chapter 2, an efficient process for calculating a viewshed for one observer is explained. In chapter 3, this calculation method is modified to efficiently process many sensors in parallel. In chapter 4, the concept of cumulative probability is introduced, which leads to a new metric for measuring the effectiveness of a collection of observers. In chapter 5, methods for optimizing this new metric are introduced. In chapter 6, results of optimization on real-life terrains are shown.

Chapter 2

VIEWSHED CALCULATION

Viewsheds in this study are calculated with the `r2` algorithm. Each observer o has a 2D geographical coordinate $\vec{r}_o$ and a range to 50% probability of detection R, all measured in meters. A set of line segments are defined, each with length $2R$, and equally spaced to form spokes in a circular pattern, sharing $\vec{r}_o$ as a common endpoint. The number of line segments is 1.5 times the circumference of the circle so that each cell close to a line segment. Figure 2.1 shows how these line segments differ from those in `r3`, which connect o to every cell in the circle.

Define the DSM value at a cell c as g_c, a floating point value in units of meters above sea level. Each cell c on the grid has a center point location $\vec{r}_c$, and objects to detect on cell c have a constant height above ground $h_c = 1\text{m}$, which results in a

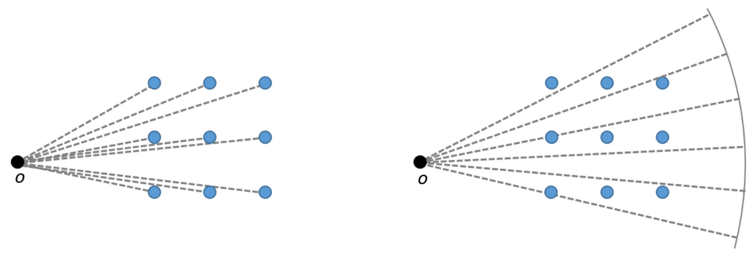


Figure 2.1. (*Left*) `r3` performs calculations on line segments drawn from o to every cell. (*Right*) `r2` only defines line segments to a set of cells on the viewshed perimeter [1]. It uses the closest line segment to each cell to approximate its line of sight.

height above sea level of $H_c = h_c + g_c$. Each observer has a height above ground level $h_o = 20\text{m}$ and a height above sea level of $H_o = h_o + g_o$. The distance between observer o and cell c is approximated as the distance between their 2D ground coordinates due to the large distances involved, i.e., $r_{o,c} = ||\vec{r}_o - \vec{r}_c||$.

Each time a line segment intersects a grid line, compute the height at the intersection point as the average DSM value of the two cells that share the grid line boundary, $[g_c + g_d]/2$. Then measure the vertical slope from H_o to this intersection point. The highest slope so far along the line segment is kept in memory.

Whenever the algorithm reaches the intersection point closest to a cell among intersections from all line segments, determine the LOS to that cell. For an object placed on cell c measure the slope as $(H_c - H_o)/r_{o,c}$. The cell is marked as visible if and only if this slope is exceeds all previously stored slopes on the line segment. The slope to c is not stored, since we do not want these potential objects to act as occlusions. Define $l_{o,c}$ as 1 if there is LOS between the o and c and 0 otherwise.

The `r2` algorithm can easily be implemented to run in parallel, since calculations for each of the line segments are independent [1]. This study does so using a GPU and CUDA C++ code.

Chapter 3

PARALLEL VIEWSHED CALCULATION FOR MULTIPLE TOWERS

When calculating multiple viewsheds, the size of the DSM data's memory depends on the range of each sensor, the terrain area in range of any sensor, the DSM's resolution, and the DSM's data type. It is possible for the necessary DSM data to take more space than GPU RAM can hold. In these instances, the DSM can be split up into a set of smaller regions, or tiles. Whichever tiles are needed to cover a sensor's viewshed are loaded into the GPU and unloaded one at a time. The order in which tiles are loaded is important in previous implementations because the `r2` algorithm calculates slopes along the line segments emanating from the sensor [1]. One way to organize the loading of tiles is to calculate a sector of the viewshed at a time, loading only the terrain tiles necessary to cover that sector. This subset of terrain tiles must not exceed the GPU RAM.

As the number of sensors increases, the number of times each tile is loaded and unloaded also increases. These transfers use up the GPU's memory bandwidth and can be very repetitive if multiple sensors need the same tile. Even individual sensors may load a tile multiple times due to the overlap between sectors. This study proposes an efficient method of carrying out the `r2` algorithm in parallel for all sensors so that each tile is used for the calculation of all viewsheds before it is unloaded. This method only needs to load and unload each tile at most twice regardless of the number of sensors. Note that with this approach, tiles may have any size, up to what can safely fit into GPU RAM, since processing will only require one tile to be loaded at a time.

The fundamental hurdle that must be overcome in order to process viewsheds in parallel is the previous strict requirement of loading enough terrain data to process each line segment all at once. Each LOS calculation requires knowledge of the largest slope encountered along the line segment up to that point. The proposed solution is to eliminate this requirement by introducing two pre-processing steps before the LOS calculation step, as shown in Figure 3.1: intra-tile slope calculation, inter-tile slope aggregation, and LOS calculation.

For initial setup, start by using the CPU to load the DSM values at each sensor location to calculate H_o, since the CPU has more RAM. These H_o values are used to calculate slopes from each sensor to various terrain locations.

3.1 Intra-tile Slope Calculation

Consider a set of line segments emanating from each sensor, just as in the `r2` algorithm. The first stage iterates through every tile in the area being studied and calculates which line segments intersect each tile. Tiles whose boundaries are not in range of any sensor can be quickly skipped over in this stage. For tiles that are not skipped, their DSM data are loaded, and the line segments are processed in parallel, with no particular need to group line segments based on their original sensors. Each tile is processed in series, which allows the GPU to load and unload DSM data.

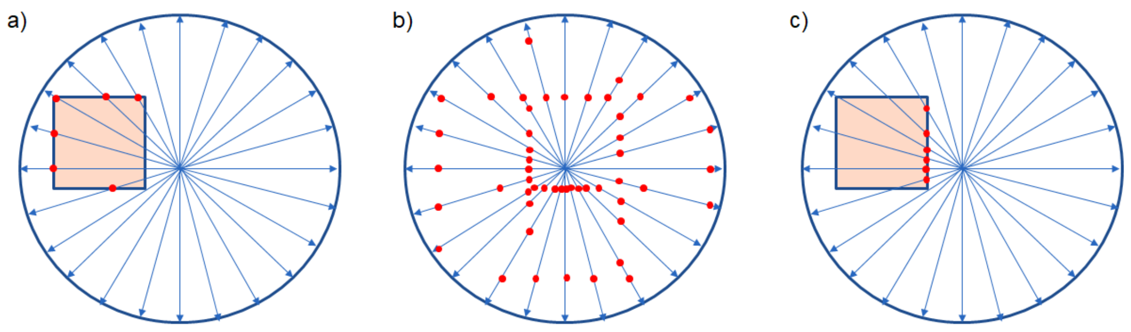


Figure 3.1. Parallel viewshed calculation procedure, where one sensor is shown for simplicity. Arrows represent the line segments emanating from the sensor as well as the direction of calculation. The orange square is the current tile being processed. Red dots indicate slope measurements relevant to each step. In a), the largest slopes within tiles are calculated for all line segments. In b), slope data among all tiles are reorganized such that each tile now contains the largest slope on each line segment encountered before it intersects the tile. In c), viewsheds are calculated on a tile by tile basis thanks to the tiles' knowledge of previous slopes.

For each line segment, calculations proceed much as they would in the slope calculation step of the `r2` algorithm, where the line segment is followed in an outward direction from the sensor. Starting and ending at the points where the line segment intersects the tile, the vertical slope from its sensor to the ground is calculated at each cell intersection, and the maximum slope is stored. The result of this stage is a large multi-dimensional array in GPU memory holding the maximum slope encountered along each line segment in each tile.

3.2 Inter-tile Slope Aggregation

The maximum slope along each line segment is now known on a tile-by-tile basis, but we need to propagate this information between tiles. Instead of having each tile record the largest slope on each line segment encountered within the tile, we would like to record the largest slope encountered before reaching the tile if following line segments outwards from the sensor.

The slope aggregation phase requires no DSM data, only slopes recorded in all tiles, so it is a short step. Each line segment has a set of slope measurements S, where S_i is the slope on each tile starting from the one that contains the sensor and moving outward. S_1 is unchanged. We then calculate the maximum slope before reaching i as

$$T_i = \max_{j<i} S_j,\ i > 1, \tag{3.1}$$

and store these T values in the multi-dimensional array for use in the last step.

3.3 LOS Calculation

For the LOS calculation step, each tile is again loaded, processed, and unloaded one at a time. Since each line segment in the tile has knowledge of T, the highest slope encountered before reaching the tile, there is no need to load any other tiles. For each line segment within the tile, LOS calculation proceeds exactly as it would in `r2`, with the exception that there is a pre-existing maximum slope value T. That is, slopes are calculated from the sensor to the ground within the tile again, storing the maximum value. During this process, line segments that pass closest to a cell's center undergo an LOS check, where an object placed on that cell is visible if the slope from the sensor to the object at height H_c is above the current maximum. The final 0 and 1 LOS values for cells within this tile are transferred back to the CPU, where they are mapped to their respective positions within their sensors' viewsheds.

3.4 CUDA Architecture

In the slope calculation and LOS calculation steps, each line segment's calculations are handled by one thread, since they are independent of each other. Threads are organized into blocks that each contain a certain number of threads, not necessarily all from the same sensor. Blocks are organized into one grid containing all line segments for all observers.

Global memory is used to store the DSM data for one tile at a time, the slope information for each tile, and the final LOS results for one tile at a time. Constant memory may be used to hold sensor attributes such as H_o and R for a slight performance boost.

Chapter 4

COMPARING COVERAGE

Let $P_{o,c}$ denote o observer's probability of detecting a previously undetected object on cell c. This probability clearly depends on their distance $r_{o,c}$. In general, there is no closed-form equation for $P_{o,c}$ that encapsulates the full nature of this relationship, since it can also depend on factors such as sensor or transmitter modalities and specifications, atmospheric turbulence, human or machine interpretation of readings, etc. For simplicity and flexibility, $P_{o,c}$ is often modeled with a logistic curve following [6],

$$P_{o,c} = \alpha\, l_{o,c}\left[1 - \frac{1}{1 + e^{-\beta(r_{o,c}-R)}}\right]. \tag{4.1}$$

The constant β controls the sinuosity of the curve, and α is a scaling factor, and R is a characteristic distance. The detection probability is 0 if there is no LOS, i.e., $l_{o,c} = 0$. At a distance $r = 0$, $P_{o,c}$ is at a maximum with constant value α, while at $r = R$, $P_{o,c} = \alpha/2$. At a distance R, this curve reaches an inflection point where $P_{o,c}$ falls most rapidly as a function of the distance. As the distance approaches infinity, $P_{o,c}$ asymptotically goes to 0.

The value of α can be any constant between 0 and 1, depending on the application. It can be thought of as a way to account for the detection of objects that move, or where persistence is sometimes an issue. Certain sensors such as pan/tilt/zoom enabled cameras can only see areas within their field of view (FOV) at any one time, but this is just a small portion of all the areas it could potentially see via panning and zooming, called its field of regard [2]. The FOV can move over time, possibly in a repetitive pattern, as well as change size depending on the zoom level. Here α can roughly approximate an average probability that during the time the object is in cell c, the FOV contains that cell as well, assuming the object moves at a constant speed. For radio or cell communication towers, or for omni-directional sensors such as some radar towers, this may not be a concern. In many cases though, setting $\alpha < 1$ can be used to create an incentive to cover multiple cells along an object's movement path, which will be described in the path viability section.

Another reason to use $\alpha < 1$ is to create an incentive for multiple observers to cover the same location for redundancy in case of malfunctions or unfavorable weather conditions affecting one or more observers. In other words, each sensor could have a duty cycle with an average uptime.

We are studying the effect of changing the locations for an set O of observers, $\boldsymbol{X} = \{\vec{r}_o : o \in O\}$, hence the quantities in the following subsections are all are functions of $\boldsymbol{X}$. When combining the independent $P_{o,c}$ probabilities of all observers (e.g., from each tower) we can write the probability of any observer detecting an object in cell c as

$$P_c = 1 - \prod_{o \in O} (1 - P_{o,c}) \,. \tag{4.2}$$

4.1 Probability of Detection along a Path

Imagine an object that moves (say at a constant speed) along a path $\mathcal{P}$ of cells. The probability of detecting the object in any one cell c is assumed to be independent of the probability of detecting it in any other cell. Defining the set of cells in $\mathcal{P}$ as $C_{\mathcal{P}}$, the probability of the object being detected while following path $\mathcal{P}$ from start to finish is

$$P_{\mathcal{P}} = 1 - \prod_{c \in C_{\mathcal{P}}} (1 - P_c) \,. \tag{4.3}$$

Taking the negative logarithm of the probability of non-detection along $\mathcal{P}$ yields a simple sum

$$w_{\mathcal{P}} = -\ln(1 - P_{\mathcal{P}}) = \sum_{c \in C_{\mathcal{P}}} \rho_c \tag{4.4}$$

with

$$\rho_c = -\ln(1 - P_c) = -\sum_{o \in O} \ln(1 - P_{o,c}) \,, \tag{4.5}$$

which again is a function of $\boldsymbol{X}$.

4.2 Proposed Objective Function: Total Path Viability

For scoring a collection of observers, we consider a set of start points in the terrain, e.g., a security perimeter, and a set of end points, which could represent precious infrastructure. The ideal case would be that there are no paths an object could take from any start point to any end point where the probability of being detected by an observer along the way is negligibly low. Our objective function will quantify how close a real scenario is to the ideal case.

We introduce a graph whose nodes represent cells c and the edges connect each cells to its adjacent cells, up to four for a rectangular grid. Every path defined so far is a sequence of these edges, which includes a corresponding list of cells. In addition to the nodes representing the cells of the terrain, we create two virtual nodes s and t that connect to all start nodes and all end nodes, respectively. Naturally, $P_s = P_t = 0$ since those nodes do not correspond a physical cell on the terrain. This appears to introduce some constraint on the methods applicability if the graph is to remain planar; namely, that the start and end nodes must lie on the outer face[1] of the graph and be arranged such that their edges with s and t, respectively, do not overlap. While this is mathematically important, it is not limiting in typical scenarios.

We consider every st-path $\mathcal{P}_{st}$ whose endpoints are s and t. A property of each edge e is the maximum probability that an object *will not* be detected along an st-path that includes e, called its viability v_e. To calculate this, first let $\mathcal{P}^*_{st,e}$ be the "shortest" st-path via edge e such that it has the minimum $\sum \rho_c$. An efficient method to calculate $\mathcal{P}^*_{st,e}$ involves creating a temporary new graph, which is the same as the original graph except that each edge cd is removed and replaced by a node $\hat{e}$ and two edges, $c\hat{e}$ and $\hat{e}d$. The new edges are given weights

$$w_{c\hat{e}} = \frac{\rho_c}{2}\,, \quad \forall c \in C_{\mathcal{P}_{st,e}} \tag{4.6}$$

which avoids double counting after doubling the number of edges on the graph. With these weights, Dijkstra's algorithm is used to calculate the shortest path $\mathcal{P}^*_{s\hat{e}}$ from s to every newly created node $\hat{e}$, giving the total weight of $\mathcal{P}^*_{s\hat{e}}$. The same procedure is repeated for shortest paths from t, giving weights for paths $\mathcal{P}^*_{\hat{e}t}$. Their sum is equivalent to $w_{\mathcal{P}^*_{st,e}}$. The viability of e requires converting from logarithms back to a probability:

$$v_e = \exp\left(-w_{\mathcal{P}^*_{st,e}}\right). \tag{4.7}$$

After calculating v_e, the auxiliary graph is no longer needed, so we will refer back to the original graph. The runtime could be improved down to linear time by using a different, more complex algorithm that takes advantage of the planar nature of the graph [9].

Let us define an st-cut as a set of edges that, if removed from the graph, would separate our graph into two connected sub-graphs; one containing s and the other

[1]The outer face is the infinitely large region outside of the terrain that is bounded by edges.

containing t, which matches the ideal case. The min cut is the minimum weight st-cut, and it is our objective function value. Finding the min cut is a max flow min cut problem [9]. In order to formulate this problem, each edge's weight must be written as a capacity.

Start with a simple case where the capacity q_e of edge e is 1 if its viability v_e is above some threshold probability and 0 otherwise. If and only if $q_v = 1$, we know that it is part of an st-path where the probability of detection is below that threshold. Call such a path a viable path. In fact, the set of edges where $v = 1$ is the set of edges that make up all possible viable st-paths. Each edge can have a "flow" that goes through it that cannot exceed the capacity. For problems where capacities and flows are restricted to 0 and 1, the max flow is the maximum number of disjoint viable st-paths, i.e. sharing no edges, that can exist in the graph. The max flow min cut theorem states that the max flow equals the min cut [9], meaning that the min cut includes exactly one edge from each disjoint viable st-path, and the total count is indeed the desired objective for this case.

Returning to the full problem, we would like to not simply find the maximum number of disjoint viable st-paths, but instead find the maximum sum of viability scores. Thus, we can instead use v_e as the capacity directly, as the viability score gives each edge knowledge of the flow that can pass through it. The st-paths included in the min cut are not guaranteed to be disjoint when flows and capacities are not integer numbers, but the min cut does consist of the maximum total viability score, which is the desired property of the objective function.

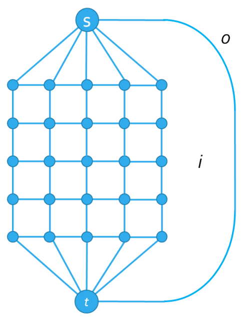
Since this graph is planar and undirected, the max flow min cut problem can be solved by constructing a dual graph as in [10]. This study's graph is a special case due to the fact that nodes s and t must lie on the outer face, which allows a single shortest path calculation to solve the problem. The simplification involves first creating another edge st with capacity 0 that still maintains the planarity of the graph. The new edge creates two new faces, called i (inside) and o (outside). Face o is infinitely large.

Next, compute the dual graph, which converts faces into nodes and nodes into faces, as Figure 4.1 shows. Edge capacities are preserved, but edges are re-routed to connect to new dual nodes. The nodes in the dual graph created from faces i and o are i' and o'. According to [11], the min st-cut through a planar graph is the same as the lowest weight cycle in its dual graph that surrounds the finite face t'. Edge $i'o'$ is necessarily a part of the lowest capacity cycle, but it has a capacity of 0, so K is the lowest weight $i'o'$-path, called $\mathcal{P}^*_{i'o'}$. Its total weight is our objective function f.

Dijkstra's algorithm is used for this step as well, such that the total time to calculate f is $O(n \log n)$. It is possible to use A^*, or best-first search with a heuristic, for faster practical performance, but the bottleneck is in the calculation of v_c since that requires the shortest path from one node to many. It is also possible to use a linear time shortest path algorithm as in [9]. To summarize, the objective f, representing the total path viability, can be written as

$$f(\boldsymbol{X}) = \sum_{e \in \mathcal{P}^*_{i'o'}} v_e(\boldsymbol{X}). \tag{4.8}$$

The best possible f score is 0. Since $v_e \leq 1$, the worst possible score is the maximum possible number of disjoint st-paths, which is bounded by the number of start nodes or end nodes, whichever is smaller. The total viability function f can also be thought of as the amount of missing coverage that could be added to individual cells via security guards, inexpensive short range technologies, etc., in order to completely deter movement from the start cells to the end cells, i.e., the ideal case.



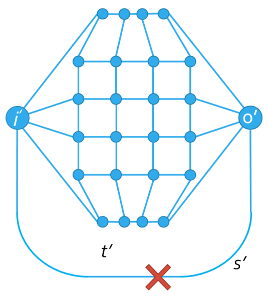


Figure 4.1. (*Left*) Graph of the problem with an extra 0-capacity edge st, which creates faces i and o. (Right) Dual Graph of the problem. The dual of edge st is $i'o'$, which separates faces s' and t'. Since $i'o'$ has a capacity of 0, it can be ignored.

Chapter 5

OPTIMIZING COVERAGE

Now we will explain methods to minimize f with respect to $\boldsymbol{X}$, with the constraint that $\vec{r}_o$ for each observer is a coordinate that is inside the terrain grid. Although the components of $\boldsymbol{X}$ are only allowed to take on integer cell values, because of the large number of cells, continuous optimization algorithms is used instead of discrete optimization. For the examples in this study, the start nodes correspond to the top row of cells and the end nodes correspond to the bottom row of cells. All code for this study is written in C++.

A small test terrain illustrates the optimization algorithms in non-convex problems where the global minimum can readily be found. This terrain consists of a flat 256×768 cell rectangular region, on which lie two square pyramids that are slightly offset from the center so that a tower can see farther from the top of the right pyramid compared to the left pyramid. Each cell is 5m across. The sides of the pyramid are at a 21° angle from the ground so that towers can see the whole pyramid if positioned close to the top. The values of the constants used to calculate $P_{o,c}$ are $\alpha = 0.03$, $\beta = 0.002$, $R = 1.38$km. Figure 5.1 shows optimization results for four different optimizers.

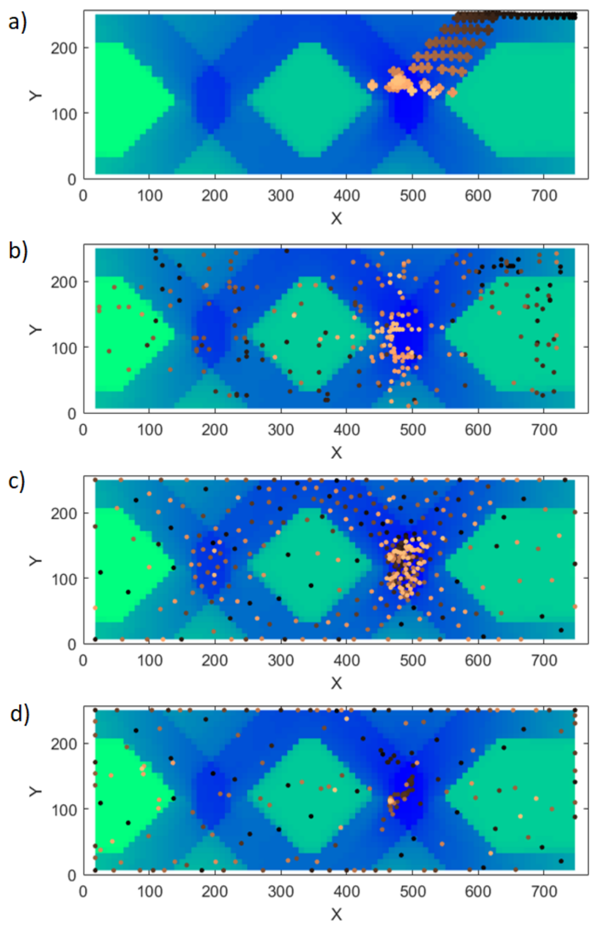


Figure 5.1. Tower locations within the two pyramids terrain are shown with points from the beginning of the optimization (*black*) until the end (*gold*). In the background is a set of f measurements along a grid, with values ranging from 150 (*blue*) to 600 (*green*). The global minimum is any tower location in the vicinity of point (500, 128). The panels illustrate different strategies (from *top* to *bottom*): Gradient Descent, SA, Student's-t BO, and Gaussian BO.

5.1 Gradient Descent

As a basic benchmark for comparison purposes, Gradient Descent (GD) is shown as the first optimizer in Figure 5.1. It is suitable if a specific starting location is picked, which in this case was the top right corner. From this starting position, GD arrives at the global minimum, where a tower is placed in the vicinity of the peak of the right pyramid.

In general terrains, Gradient Descent is not suitable because there could be a large number of local minima. Additionally, any optimization method that relies on gradients is generally not preferred for two reasons. First, there is no closed form equation for the gradient of this objective function, so a gradient would need to be empirically calculated as a finite difference between multiple objective measurements along each dimension, which is expensive. Second, as will be shown later in a more complex terrain, f can sometimes vary by a large amount when moving a tower by a small distance, making gradients hard to rely on for optimization.

5.2 Simulated Annealing

The next optimizer is Simulated Annealing (SA), a stochastic optimization algorithm [6]. It has an internal property called temperature that determines the probability of moving to a parameter set with a higher f value in order to escape local minima. The temperature starts at a high initial value and decreases exponentially over time, so the optimizer is initially very willing to move around the parameter space and leave local minima it encounters, but eventually settles in an area close to a local minimum.

The `ensmallen` library provides a C++ SA implementation with many tuning parameters [12]. It also provides a method for choosing the next parameter set in each iteration using feedback control. The user supplies three parameters to the feedback controller: an initial move distance, a maximum allowable move distance, and a feedback gain between 0 and 1. In each iteration, the optimizer randomly increases or decreases only one parameter $\boldsymbol{X}_i$ by the calculated distance. Since this library does not take in upper or lower bounds for $\boldsymbol{X}_i$, $f(\boldsymbol{X})$ is modified to immediately return large values for values of $\boldsymbol{X}$ outside of the grid.

The SA optimizer finds the global minimum in the pyramid terrain, and a cluster of recent measurements can be seen in 5.1. SA tends to move the tower by only short distances as the temperature drops at the end of the run, which is why most of the lighter dots are clustered together. Some straight horizontal and vertical patterns can be seen with this optimizer because only one dimension can change from each iteration to the next.

5.3 Bayesian Optimization

Bayesian Optimization (BO) treats the objective function f as a black box while balancing exploration and exploitation of the parameter space [13]. Also called Kriging, it is most useful in cases where the objective function is relatively expensive to calculate, as it takes time for BO to estimate the best parameter set to measure at each iteration. BO is most effective when the number of dimensions is 20 or lower [13], so the numbers of towers and parameters per tower are kept relatively low in this study.

BO works by keeping track of a surrogate model for f that is less expensive to evaluate. A stochastic function, commonly a Gaussian Process, is chosen as the surrogate in order to represent the optimizer's uncertainty about the true form of f. Every evaluation of f is stored as data D such that the surrogate model serves as a posterior $P(f|D)$. After some initial seed evaluations, each iteration consists of using the previous $P(f|D)$ as a prior, deciding the next parameter set to evaluate, incorporating the evaluated result into D, and updating the posterior. This process continues until a set number of iterations elapses, meaning that unlike SA, it is possible for the parameter set to change drastically even far into the optimization run.

An open source C++ package called BayesOpt [13] is used to perform Bayesian optimization. There are several settings to select when using BayesOpt. This study uses two different surrogate models, Student's-t process and Gaussian process. The acquisition function is used to determine the next objective function evaluation's parameters. This study uses Expected Improvement (EI), which chooses the parameter set with the highest expectation value of f reduction compared to the best value obtained so far. The parameter sets for initial seed evaluations are selected using Latin Hypercube Sampling, a quasi-random method that covers the multidimensional parameter space relatively more evenly than uniform random sampling [13]. The number of initial evaluations vary based on the complexity of the problem.

Optimization runs in the pyramid terrain for BO with Student's-t surrogate (Student's-t BO) and BO with Gaussian Process surrogate (Gaussian BO) are shown in Figure 5.1. Both optimizers find the global minimum shortly after the initial evaluations, but their behaviors after that point differ. Gaussian BO favors more exploration, filling up the space and spending more iterations on the edges, while Student's-t BO exploits the area around the global minimum, as seen by the large cluster of measurements there.

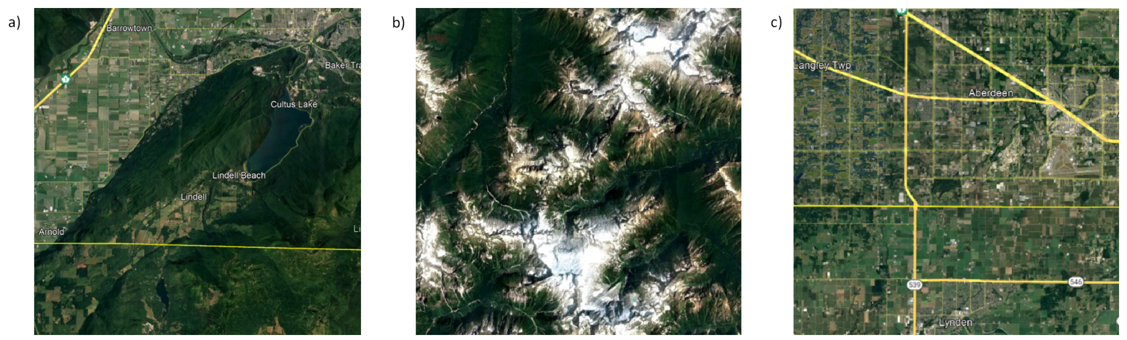


Figure 5.2. Satellite imagery of the three terrain regions used in this study, all east of Vancouver, Canada. a) Rural and mountainous terrain with a ridge and a lake. b) Extremely mountainous terrain. c) Relatively flat terrain with trees and buildings.

Chapter 6

OPTIMIZING ON REAL TERRAINS

The dataset to calculate realistic viewsheds is a 5m resolution DSM collected by LiDAR scans. Three geographic areas in the Cascade mountain range near Vancouver, Canada are chosen due to the variety of topographies in that region. These approximate areas are shown in Figure 5.2. Each area is square measuring $4,096$ cells or around 20.5km on each side.

6.1 Single Tower: One Dimension

For optimization runs using real DSM data, the probability of detection constants are $\alpha = 0.03$, $\beta = 0.004$, $R = 3.75$km. To get a general sense of the shape of the objective function, a simple optimization of one tower's location in 1 dimension within the valley terrain is shown in Figure 6.1, keeping the y-cell coordinate fixed at $y = 2,148$. Student's-t BO finds the global minimum well within the allotted 100 iterations, and there is a high density of points with x-values close to the global minimum

From the brute force scan of f along this line, some properties of f are evident. In some areas, f is quite noisy and jagged, with some peaks and valleys under 10

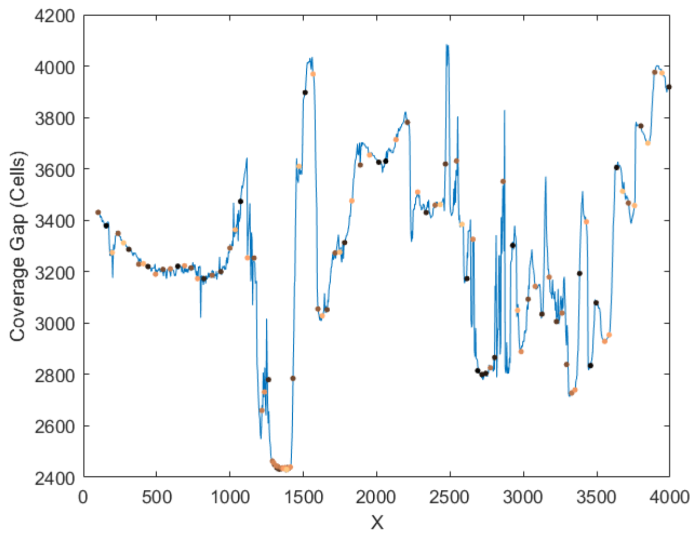


Figure 6.1. A 1D optimization run in the valley terrain with the tower's y-coordinate fixed to 2,148. Points are colored from oldest (*black*) to newest (*gold*). The optimizer is Student's-t BO with 110 iterations. The blue line consists of f measurements evenly spaced 4 cells apart.

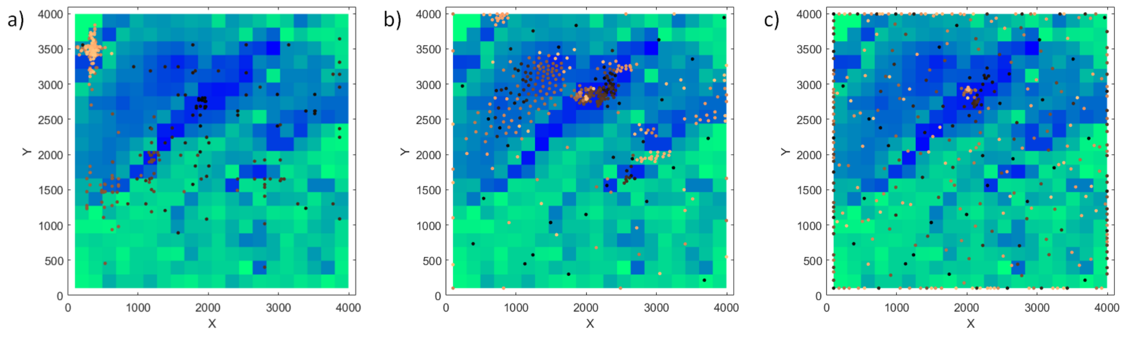


Figure 6.2. Optimization runs placing one tower in the valley terrain, where points are colored from oldest (*black*) to newest (*gold*). The grid scan of f in the background ranges from 2,000 (*blue*) to 4,096 (*green*). Different optimizers were used: a) SA, b) Student's-t BO, and c) Gaussian BO.

cells wide. These could be locations where moving the tower from one side of a LOS obstruction to the other side can dramatically change the viewshed, even with a small displacement. The large changes from cell to cell may pose a significant challenge for optimizers that depend on gradient information. Another feature is that the highest and lowest x-coordinates tend to produce higher f values than their neighbors farther from the edges, because areas outside the rectangular terrain are not counted as detectable. These properties also apply in the case of optimizing multiple towers' coordinates.

6.2 Single Tower: Two Dimensions

Defining the set of best tower locations as $\mathbf{X}^*$, Figure 6.2 shows the three optimizers' results with two spatial dimensions for each observer. Each optimizer is given a maximum of 375 iterations. SA and Student's-t BO, and Gaussian BO all have similar $f(\mathbf{X}^*)$ values: 2,425, 2,486, and 2,490 respectively, although the corresponding tower locations differ, showing that they find different local minima. All three optimizers reach a local minimum quickly, within the first 100 iterations, and improve very little from there. The minimum value of the brute force grid scan is 2,366, which is only slightly lower than the optimizer results, indicating that optimizing one tower's coverage is not very difficult in this terrain, and that there are many similarly good placement locations.

SA's graph has a dense cluster of recent points and a cloud of older points. Some of the old points also form small clusters in areas of low f near the center ridgeline. These clusters indicate times when SA finds local minima, breaks away to continue searching, and finally settles in a local region in the northwest corner as the temperature approached 0. Student's-t BO's measurements show tight clusters both old and recent, with perhaps the strongest preference towards exploitation vs. exploration among the optimizers. A pseudo-random search pattern emerges in the valley west of the ridge, where scores vary gently and smoothly. Student's-t BO jumps around the map even near the end of the optimization, finally settling in the ridgeline. Gaussian BO's measurements are less clustered than those of other methods, but most of the map is covered, indicating a strong preference for exploration. There is one cluster of points on the ridgeline, where the best score is found. Gaussian BO often measures the edges

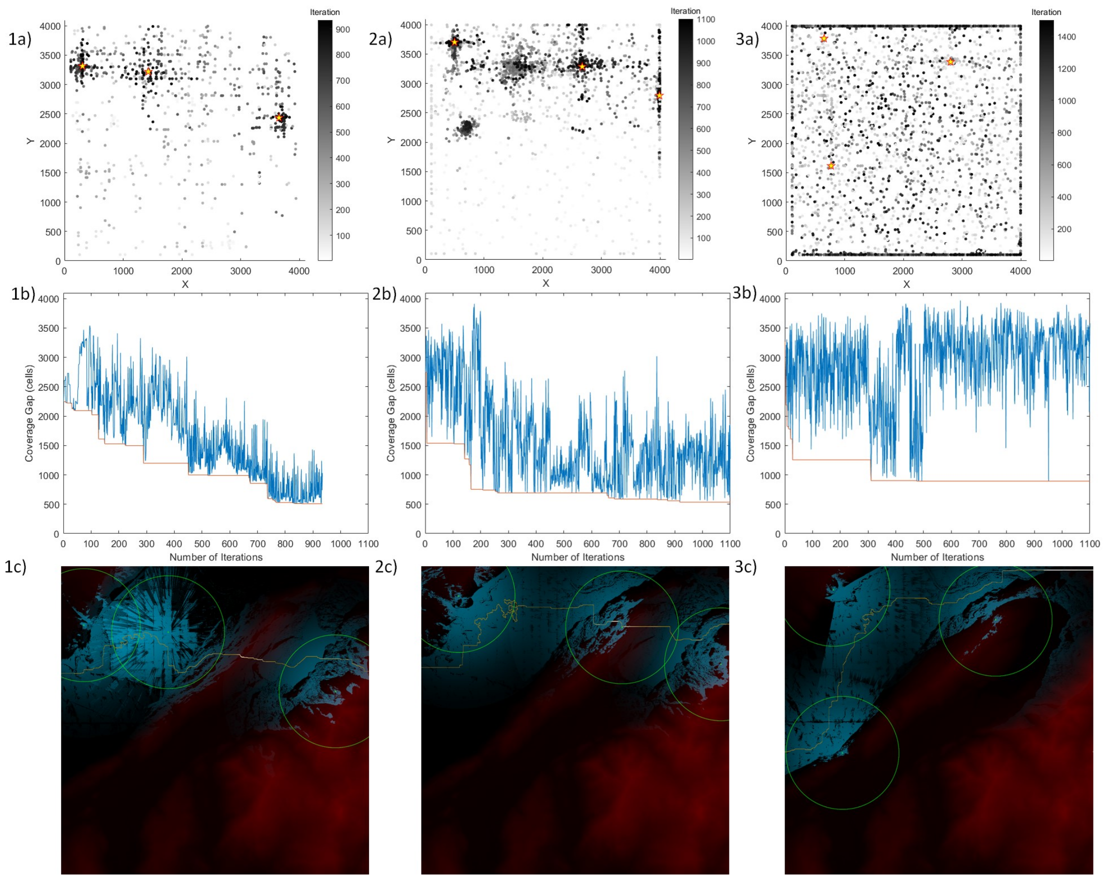


Figure 6.3. Optimization runs placing three towers in the valley terrain. 1a-3a) from left to right show tower locations at each iteration for SA, Student's-t BO, and Gaussian BO. Yellow stars show $\mathbf{X}^*$, the set of tower locations with minimum f. 1b-3b) show current and minimum f vs. iteration. 1c-3c) show the towers positioned at $\mathbf{X}^*$. Bluer regions indicate higher P_c and redder regions indicate higher elevation. Green rings are drawn at a radius R around each tower. The min cut's path $\mathcal{P}^*_{i'o'}$ is drawn in gold.

of the terrain, which may be a result of maximizing expected improvement over a constrained parameter space.

6.3 Multiple Towers

Next, three towers are to be placed on the terrain, increasing the total number of dimensions in the parameter space from 6 to 9. Each tower has the same probability of detection constants, although in general this is not necessary. Each optimization run is shown in Figure 6.3. The number of iterations is increased to 1,100, including 200 initial iterations. The cooling function of SA has a corresponding cooling rate so that the temperature function approaches 0 approximately at the end of these iterations.

Once again, the behavior of SA and Student's-t BO show several similarities, including similar scores of 505 and 535, respectively. Over the length of the optimization runs, their scores trend downward and never return to their original high values. Both optimizers use two towers to cover the valley and one to cover the lake from the east side, leaving only small coverage gaps in the center ridgeline. The exact positions of the three towers differ though. It also appears that Student's-t BO spends some iterations with a tower near the SA solution's middle tower, but eventually abandons that location.

On the other hand, Gaussian BO spends a large portion of its run without making improvements to its best f score, 892. Additionally, there are many high f values even late into the optimization process. Its focus on exploration, as can be seen by the lack of clustering, may not scale well with the number of dimensions in the parameter space. Its best tower configuration shares one tower location with that of Student's-t BO, while the two western towers appear to be placed on ridges on either side of a valley, leaving a coverage gap at the northeast corner.

In this more realistic scenario, Figure 6.4 visualizes v_e and shows that path viability can act as a kind of "filter" for P_c values with two important effects. First, v_e is high for cells surrounded by a region of cells with high P_c, even if the cell itself has low P_c. This means: 1) the "thickness" of coverage contributes to f, and 2) if an area has low coverage, but is totally surrounded by areas with high coverage, then it does not count as a coverage gap since an object cannot viably travel from s to t through the area. Several speckles of areas lacking coverage in the northwest corner disappear in the v_e graph following this principle. Second, "cracks", or long and narrow regions with low P_c, are maintained in v_e, highlighting areas with coverage gaps.

Optimization can of course include more than 3 towers. Figure 6.5 shows the

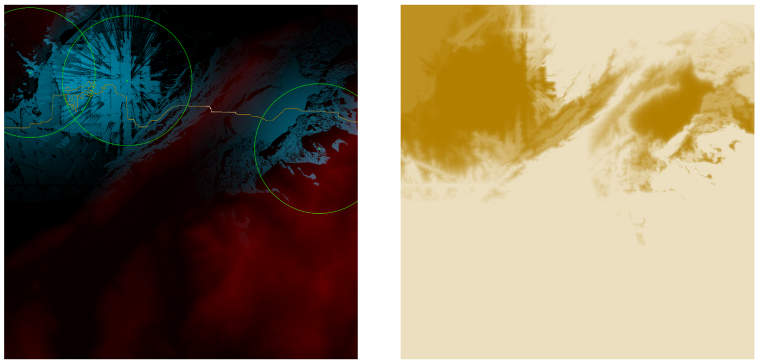


Figure 6.4. *(Left)* Towers placed on a terrain, where brighter blue indicates higher P_c and brighter red indicates higher elevation. *(Right)* Path viability through each edge, v_e, is higher for cells with lighter colors.

SA optimizer's result with twice the number of towers, each with a smaller radius of $R = 5.3km$ to keep the total area of their circles roughly the same. Just as in the 3 tower case, the towers are roughly aligned horizontally, and there is good coverage in the valley and in the lake. With a higher tower count, a pattern emerges where towers are more tightly clustered in rough terrain and more spread apart in flat, open terrain. The three towers in the western valley region cover each others' blind spots quite well. The best value found for f, 653, is somewhat higher than for the 3 tower case, with no coverage in the center ridge. It is possible that the smaller radius means that towers positioned around the center ridge cannot see as far into the ridge or have to view it at a steeper angle, resulting in less coverage of the ridge. Another possibility is that since there are more parameters to optimize, the optimizer would need more iterations in order to reach a similar score as for the 3 tower scenario, although this would be hard to compare using SA because such a change would also require changing the temperature cooling rate.

6.4 Mountainous and Rural Areas

A few additional optimization runs are showcased for other scenarios. Figure 6.6 compares the same SA optimizer's results among the three terrains. The mountainous

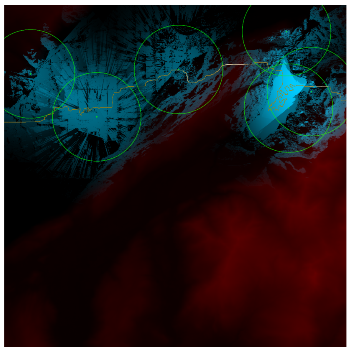


Figure 6.5. SA Optimization run with 6 towers on valley terrain, where the total area of the green rings is the same as in the 3 tower optimization run.

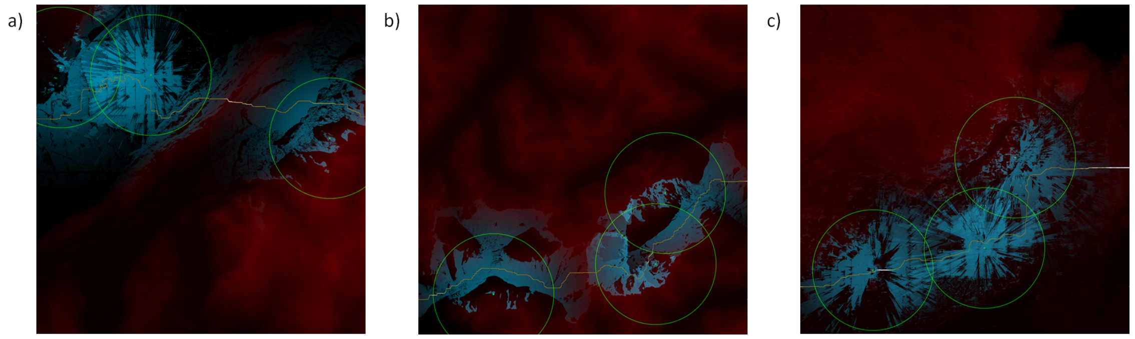


Figure 6.6. SA optimization runs on all three terrains from Figure 5.2: a) valley terrain, repeated from previous figure, b) mountain terrain, and c) rural terrain.

terrain is interesting due to its sharp mountain peaks, which result in diverse and complex viewsheds shapes. Each viewshed has large areas with continuous coverage, which may indicate a lack of trees and buildings in such a remote area location with high altitudes. The SA optimizer finds a solution with very few coverage gaps, $f = 215$. It places every tower on a mountain peak or flank, which makes sense for maximizing coverage. Together, the towers cover two valleys and both sides of a mountain ridge that separates them, forming a wide band. Interestingly, the viewshed coverage is relatively poor in the immediate vicinity around each tower, which indicates that the towers have poor visibility of their own mountain's flanks, yet high visibility to the valleys below.

The rural terrain is very flat, so terrain is not a significant LOS impediment for the towers. Instead, many small features such as trees and buildings create cuts and gaps in their viewsheds. Interleaving coverage between neighboring towers appears to be a method the SA optimizer uses to overcome this difficulty, but this need for overlap opens up a gap on the east side. As a result, the score for this terrain, 797, is noticeably higher than for the other terrains using the same SA optimizer.

6.5 Timing and Performance

Optimization runs in this study are done on a single desktop computer, utilizing its GPU for viewshed calculation and CPU for f evaluations and optimization. The viewshed calculation step does not take a significant fraction of the optimization run time for hard optimization scenarios, as the viewsheds each contain significantly fewer cells than the terrain, run in $O(n)$ time with the `r2` approximation, and are computed in parallel on the GPU. In each iteration, they typically take around 3 seconds total for the realistic terrain scenarios.

Since Djikstra's algorithm is used for shortest path calculations, f calculations are done in $O(n \log n)$ time, where the pyramid terrain contains 1.98×10^5 cells and the realistic terrains contains 1.67×10^7 cells. An array stores the graph nodes implicitly to minimize overhead as opposed to creating a `struct` for each node with pointers to other nodes. For the realistic terrain, each calculation of f consisting of three shortest path calculations takes roughly one minute.

When comparing the optimizers, BO must use the acquisition function to decide where to measure next and update the posterior of the surrogate function, requiring more overhead than SA. For the simple pyramid scenarios, this difference is negligible, but the more iterations and the more complicated the terrain, the more time this overhead takes, to the point where it is on the same order as the f calculation itself. The two surrogate functions, Student's-t and Gaussian, do not have a significant runtime difference. BO optimization runs with the full $1,100$ iterations on the real

DSM take a few days to complete.

Chapter 7

DISCUSSION AND FUTURE WORK

This study presents a novel metric with which to evaluate the placement of towers that rely on LOS to cover geographical areas. The total path viability metric is valuable for situations where the land area to cover is larger than the area that the towers can cover given range and LOS constraints. The best way to utilize the towers' scarce coverage is to form a band through the region. The number and size of coverage gaps in this band, the min *st*-cut of path viability, corresponds to areas where other types of security, like personnel, may be needed. In addition to surveillance, the notion of coverage band applies to communications coverage. Imagine that instead of using LOS coverage to deter movement from the start points to the end points, we would like to allow communications-enabled movement in a region bounded by the start and end points. Then the min cut represents a path where one could construct a trail or road that has minimal segments without signal. Thicker coverage would increase the area of coverage for offtrail movement or for infrastructure development around the trail. The signal could be one of several different types, including land mobile radio or cellular 4G/5G, although each frequency presents its own nuances in terms of ranges and types of LOS obstructions.

Optimizing our objective provides more targeted coverage than optimizing previous objectives. Figure 3 shows an optimization run in the valley terrain using the objective function in [6], which is the sum of P_c for all cells. multiplied by -1 for minimization. This *naïve* objective, $f_{\text{naïve}}$, optimizes the overall area covered, but it has no knowledge of the start or end points. Because LOS is so much worse outside the flat valley, the optimizer places all three towers in the valley. This leaves a significant coverage gap in the horizontal direction, while providing perhaps an unnecessarily high amount of coverage in the valley.

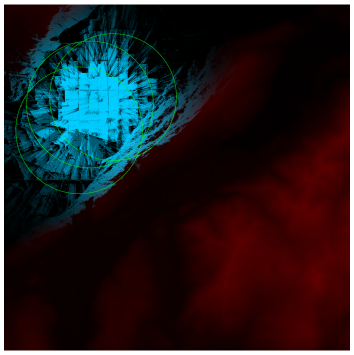


Figure 7.1. Optimization of $f_{\text{naïve}}$ using Student's-t BO with the same settings as for f_{new} on the valley terrain (Figure 6.3).

To further compare $f_{\text{naïve}}$ with our proposed objective, which we will call f_{new}, both objectives are optimized using Student's-t BO to find their best solutions, $\boldsymbol{X}^*_{\text{naïve}}$ and $\boldsymbol{X}^*_{\text{new}}$ respectively. These are then evaluated by both objective functions. We measure $f_{\text{new}}(\boldsymbol{X}^*_{\text{naïve}}) = 2,100$, which is significantly higher than $f_{\text{new}}(\boldsymbol{X}^*_{\text{new}}) = 535$. This means that, although $f_{\text{naïve}}$ is simpler and faster to calculate, it is very possible that optimizing for this objective will leave wide gaps in coverage that are problematic weaknesses for real applications. On the other hand, $f_{\text{naïve}}(\boldsymbol{X}^*_{\text{naïve}}) = -95,900$ and $f_{\text{naïve}}(\boldsymbol{X}^*_{\text{new}}) = -64,600$. While f_{new} generally does show some preference towards greater total coverage in order to cut down on the number of viable paths, it does not specialize in maximizing raw coverage, nor is it expected to. What is of more direct importance is the pattern and orientation of coverage formed by cooperating towers.

Given a suitable number of iterations, SA and Student's-t BO optimizers are able to find sensor configurations that leave only small coverage gaps. Student's-t BO is slower than SA for hard scenarios, but it does have potential advantages. It can break out of local minima and explore wide areas even towards the end of the optimization run, and its tuning parameters are less sensitive to the problem and desired iteration count. Gaussian BO is suitable for easier scenarios, but its focus on exploration does not scale well with the size of the parameter space. It is possible to improve Gaussian BO's results with a different selection of settings, but in this study Student's-t BO has better results for hard scenarios when using the same settings. Other types of optimizers could also be investigated, such as evolutionary algorithms [6].

The speed of the optimization process could be improved, which would allow for faster decision-making or more complex scenarios involving larger or higher resolution terrain areas, more observers, or more parameters per observer. One method is to implement $O(n)$ planar graph algorithms in a way that does not introduce too much overhead through recursion or pointer calls. Another option could be to set a cutoff range on each tower, set $P_c = 0$ for cells outside that range of any tower, and use this information to merge undetectable nodes, simplifying the graph. If suitable for the application, the optimizer could be run using high performance computing as well.

The performance of the parallel viewshed algorithm is surely fast enough to enable viewshed optimization. Even when placing six towers, the viewshed calculation step took just a small fraction of each optimiztion iteration's runtime. On the other hand, the speed of the optimization process could be improved, which would allow for faster decision-making or more complex scenarios involving larger or higher resolution terrain areas, more observers, or more parameters per observer. One method is to implement $O(n)$ planar graph algorithms in a way that does not introduce too much overhead through recursion or pointer calls. Another option could be to set a cutoff range on each tower, set $P_c = 0$ for cells outside that range of any tower, and use this information to merge undetectable nodes, simplifying the graph. If suitable for the application, the optimizer could be run using high performance computing as well.

There are a number of improvements that could make tower optimization more readily applicable in a real life scenario. Often times, there are significant land areas where towers cannot be placed, due to factors like restricted or unstable land, distance from power or infrastructure, etc. Restricted areas could be digitized so that those locations would not be considered in the optimization process. In a similar fashion, if there are certain areas where LOS is not necessary, such as over a body of water, those areas could be excluded from LOS calculations.

In reality, there could also exist many variable costs for placing sensors. Each location could have a cost that depends on difficulty in transporting materials for construction or maintenance. For flying sensors, it might take a variable amount of time or fuel to launch or fly to certain locations. There may also be multiple choices for sensor models, or different tower heights for stationary towers. Ideally, a budget constraint could be specified, and the optimizer could in the future be able to choose a suitable number of sensors, each with their own locations and models, such the the costs all fit within the budget. If possible, collaborating with real-life users of LOS-reliant systems can help validate or augment the optimizers.

Another avenue to explore is using different types of terrain data, such as 3D photogrammetry or computer models, which can have multiple heights in the same geographic location. Intricate geometry is especially important in urban areas, where buildings, bridges, and other structures may have multiple floors. To accommodate these data, A GPU render-based viewshed method can replace `r2` for multi-layer terrain [14]. If the terrain can no longer be represented as a planar graph, the min *st*-cut calculation may require a slower, more general method, such as the Edmonds-Karp algorithm or modern improvements to it [15].

Bibliography

[1] A. Osterman, L. Benedičič, and P. Ritoša, “An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU,” *International Journal of Geographical Information Science*, vol. 28, no. 11, pp. 2304–2327, 2014. [Online]. Available: https://doi.org/10.1080/13658816.2014.918319

[2] M. H. Friedman, “Fundamental search relationships and their application to field of regard search, search by multiple observers, search from a moving vehicle, and multitarget search,” *Optical Engineering*, vol. 52, no. 4, pp. 1 – 28, 2013. [Online]. Available: https://doi.org/10.1117/1.OE.52.4.041107

[3] Z. He, Q. Liu, X. Fan, D. Chen, S. Wang, and G. Yang, “Fiber-optic distributed acoustic sensors (DAS) and applications in railway perimeter security,” in *Advanced Sensor Systems and Applications VIII*, T. Liu and S. Jiang, Eds., vol. 10821, International Society for Optics and Photonics. SPIE, 2018, pp. 1 – 6. [Online]. Available: https://doi.org/10.1117/12.2505342

[4] A. J. Lipton, C. H. Heartwell, N. Haering, and D. Madden, “Critical asset protection, perimeter monitoring, and threat detection using automated video surveillance,” in *Proceedings of the Thirty-Sixth Annual International Carnahan Conference on Security Technology*, 2002.

[5] S. Tabik, E. Zapata, and L. Romero, "Simultaneous computation of total viewshed on large high resolution grids," *International Journal of Geographical Information Science*, vol. 27, no. 4, pp. 804–814, 2013. [Online]. Available: https://doi.org/10.1080/13658816.2012.677538

[6] V. Akbarzadeh, C. Gagne, M. Parizeau, M. Argany, and M. A. Mostafavi, "Probabilistic sensing model for sensor placement optimization based on line-of-sight coverage," *IEEE Transactions on Instrumentation and Measurement*, vol. 62, no. 2, pp. 293–303, 2013.

[7] A. A. Altahir, V. S. Asirvadam, N. H. B. Hamid, P. Sebastian, N. B. Saad, R. B. Ibrahim, and S. C. Dass, "Optimizing visual surveillance sensor coverage using dynamic programming," *IEEE Sensors Journal*, vol. 17, no. 11, pp. 3398–3405, 2017.

[8] J. Yang, M. Kamezaki, H. Iwata, and S. Sugano, "A 3d sensing model and practical sensor placement based on coverage and cost evaluation," in *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, 2015, pp. 1–6.

[9] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, "Faster Shortest-Path Algorithms for Planar Graphs," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 3–23, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000097914938

[10] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen, "Improved algorithms for min cut and max flow in undirected planar graphs," in *Proceedings of the forty-third annual ACM symposium on Theory of computing*, 2011, pp. 313–322.

[11] A. Itai and Y. Shiloach, “Maximum flow in planar networks,” *SIAM Journal on Computing*, vol. 8, no. 2, pp. 135–150, 1979. [Online]. Available: https://doi.org/10.1137/0208012

[12] R. R. Curtin, M. Edel, R. G. Prabhu, S. Basak, Z. Lou, and C. Sanderson, “The ensmallen library for flexible numerical optimization,” *Journal of Machine Learning Research*, vol. 22, no. 166, pp. 1–6, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-416.html

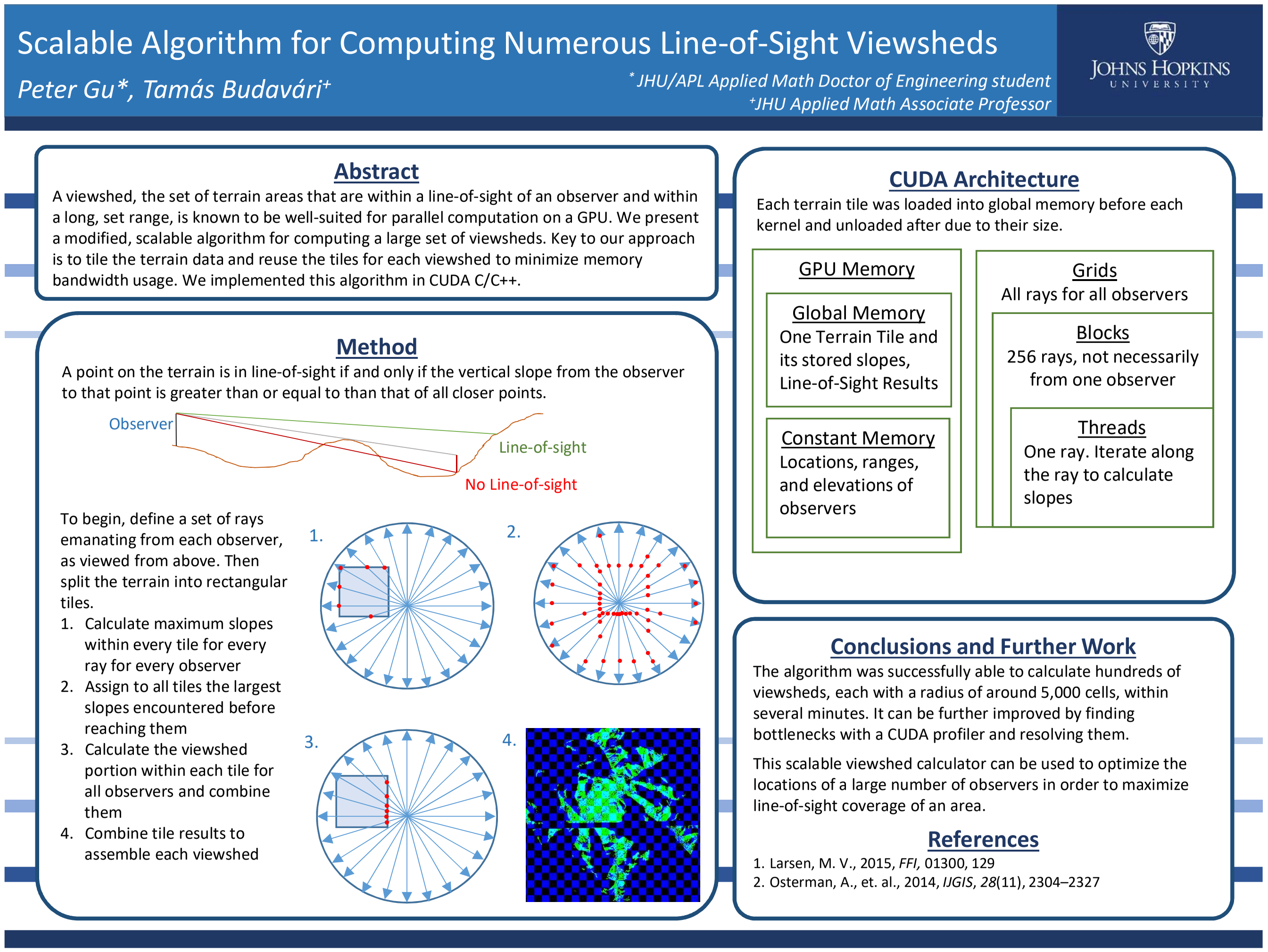
[13] R. Martinez-Cantin, “Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits,” *Journal of Machine Learning Research*, vol. 15, no. 115, pp. 3915–3919, 2014. [Online]. Available: http://jmlr.org/papers/v15/martinezcantin14a.html

[14] H. Christoph, “Gpu-based visualisation of viewshed from roads or areas in a 3d environment,” 2016.

[15] A. V. Goldberg and R. E. Tarjan, “Efficient maximum flow algorithms,” *Communications of the ACM*, vol. 57, no. 8, pp. 82–89, 2014.

Appendix I: Conference Posters

The following posters were presented online at the The Institute for Data Intensive Engineering and Science (IDIES) 2020 and IDIES 2021 Annual Symposia, hosted by Johns Hopkins University. The content in these posters is covered in the main text of this document, but the graphics can provide extra context to the methods used.



Scalable Algorithm for Computing Numerous Line-of-Sight Viewsheds

Peter Gu*, Tamás Budavári+

* JHU/APL Applied Math Doctor of Engineering student
+JHU Applied Math Associate Professor

JOHNS HOPKINS UNIVERSITY

Abstract

A viewshed, the set of terrain areas that are within a line-of-sight of an observer and within a long, set range, is known to be well-suited for parallel computation on a GPU. We present a modified, scalable algorithm for computing a large set of viewsheds. Key to our approach is to tile the terrain data and reuse the tiles for each viewshed to minimize memory bandwidth usage. We implemented this algorithm in CUDA C/C++.

Method

A point on the terrain is in line-of-sight if and only if the vertical slope from the observer to that point is greater than or equal to than that of all closer points.

To begin, define a set of rays emanating from each observer, as viewed from above. Then split the terrain into rectangular tiles.

1. Calculate maximum slopes within every tile for every ray for every observer
2. Assign to all tiles the largest slopes encountered before reaching them
3. Calculate the viewshed portion within each tile for all observers and combine them
4. Combine tile results to assemble each viewshed

CUDA Architecture

Each terrain tile was loaded into global memory before each kernel and unloaded after due to their size.

GPU Memory

Global Memory
One Terrain Tile and its stored slopes, Line-of-Sight Results

Constant Memory
Locations, ranges, and elevations of observers

Grids
All rays for all observers

Blocks
256 rays, not necessarily from one observer

Threads
One ray. Iterate along the ray to calculate slopes

Conclusions and Further Work

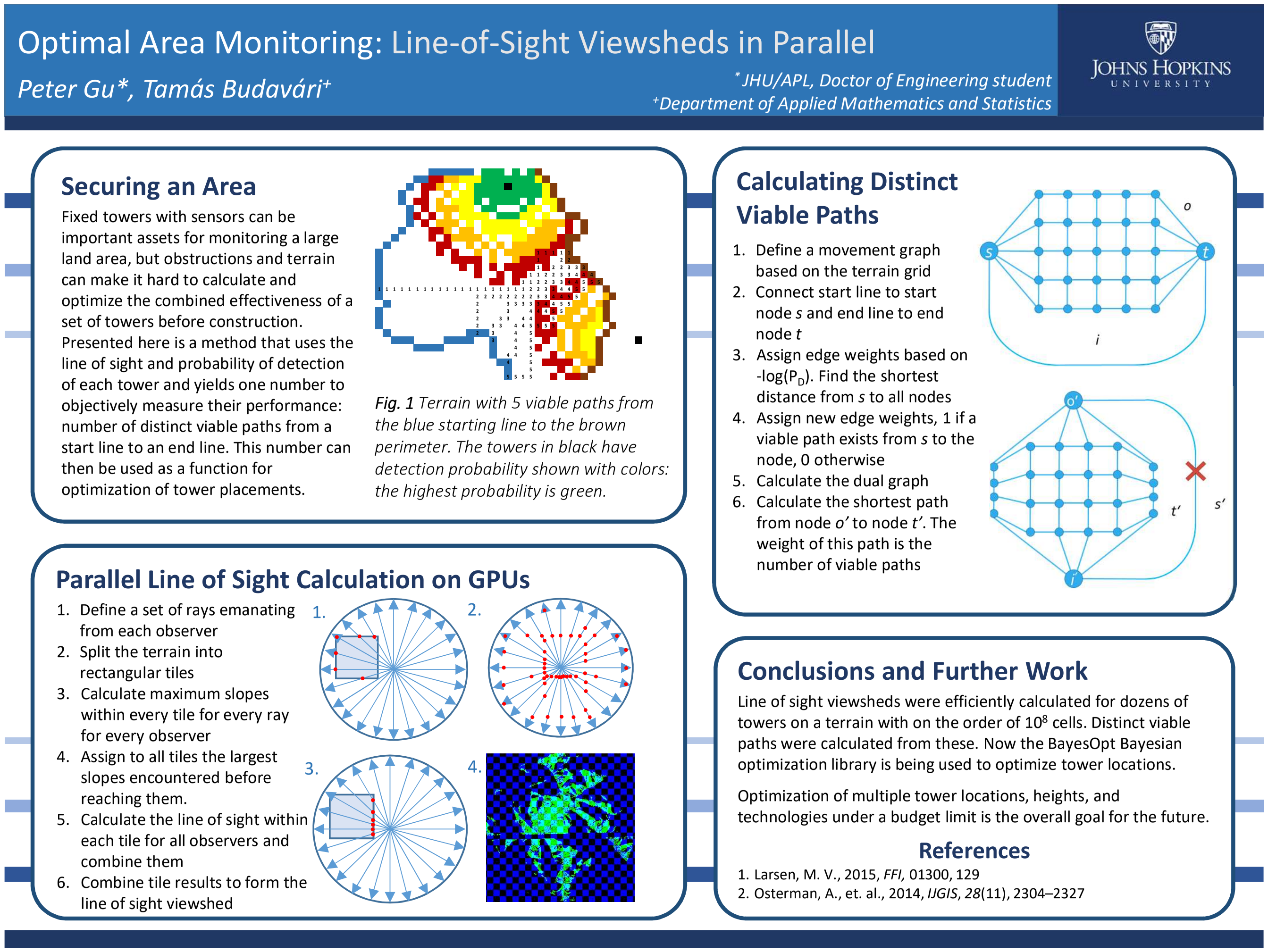
The algorithm was successfully able to calculate hundreds of viewsheds, each with a radius of around 5,000 cells, within several minutes. It can be further improved by finding bottlenecks with a CUDA profiler and resolving them.

This scalable viewshed calculator can be used to optimize the locations of a large number of observers in order to maximize line-of-sight coverage of an area.

References

1. Larsen, M. V., 2015, *FFI*, 01300, 129
2. Osterman, A., et. al., 2014, *IJGIS*, *28*(11), 2304–2327

Figure 2. Poster presented at the IDIES 2020 Annual Symposium's online poster session, focusing on LOS calculation.



Optimal Area Monitoring: Line-of-Sight Viewsheds in Parallel

Peter Gu*, Tamás Budavári+

*JHU/APL, Doctor of Engineering student
+Department of Applied Mathematics and Statistics

JOHNS HOPKINS UNIVERSITY

Securing an Area

Fixed towers with sensors can be important assets for monitoring a large land area, but obstructions and terrain can make it hard to calculate and optimize the combined effectiveness of a set of towers before construction. Presented here is a method that uses the line of sight and probability of detection of each tower and yields one number to objectively measure their performance: number of distinct viable paths from a start line to an end line. This number can then be used as a function for optimization of tower placements.

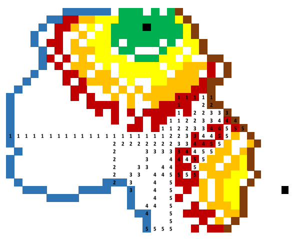
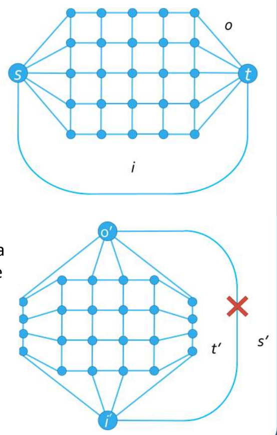


Fig. 1 *Terrain with 5 viable paths from the blue starting line to the brown perimeter. The towers in black have detection probability shown with colors: the highest probability is green.*

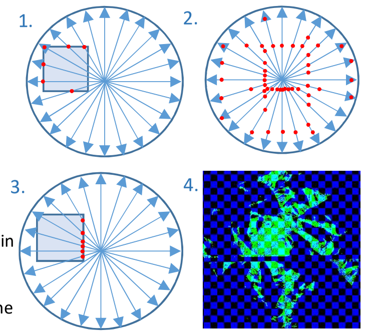
Calculating Distinct Viable Paths

1. Define a movement graph based on the terrain grid
2. Connect start line to start node *s* and end line to end node *t*
3. Assign edge weights based on -log(P_D). Find the shortest distance from *s* to all nodes
4. Assign new edge weights, 1 if a viable path exists from *s* to the node, 0 otherwise
5. Calculate the dual graph
6. Calculate the shortest path from node *o′* to node *t′*. The weight of this path is the number of viable paths



Parallel Line of Sight Calculation on GPUs

1. Define a set of rays emanating from each observer
2. Split the terrain into rectangular tiles
3. Calculate maximum slopes within every tile for every ray for every observer
4. Assign to all tiles the largest slopes encountered before reaching them.
5. Calculate the line of sight within each tile for all observers and combine them
6. Combine tile results to form the line of sight viewshed



Conclusions and Further Work

Line of sight viewsheds were efficiently calculated for dozens of towers on a terrain with on the order of 10^8 cells. Distinct viable paths were calculated from these. Now the BayesOpt Bayesian optimization library is being used to optimize tower locations.

Optimization of multiple tower locations, heights, and technologies under a budget limit is the overall goal for the future.

References

1. Larsen, M. V., 2015, *FFI*, 01300, 129
2. Osterman, A., et. al., 2014, *IJGIS*, *28*(11), 2304–2327

Figure 3. Poster presented at the IDIES 2021 Annual Symposium's online poster session. It describes the novel LOS calculation method and the new objective function.

Appendix II: Codebase Documentation

The software needed to complete this study consists of a relatively large body of newly written code and several open source libraries, all in C++. Please contact Peter Gu at pgu3@jh.edu to inquire about the source code.

A. External Libraries

- `bitmap_image.hpp`: used to create BMP image files of optimization results at each iteration. `https://github.com/ArashPartow/bitmap/blob/master/bitmap_image.hpp`
- CUDA: used to interface with GPU devices to write parallel, device level code. `https://developer.nvidia.com/cuda-downloads`
- Ensmallen: contains numerous optimization methods, out of which Simulated Annealing (SA) and Gradient Descent (GD) are used in the code [12]. `https://ensmallen.org/`
- BayesOpt: performs Bayesian Optimization with several built-in settings that can be modified [13]. `https://github.com/rmcantin/bayesopt`

7.2 Code Structure

- `OptimizationRunner.cpp` is contains the `main` method. It controls the optimization run overall, with a choice of optimization methods or brute force evaluation patterns.

- `EvaluateObjective.cpp` handles all the actions for one optimization iteration, including reformatting the sensor locations, calling the viewshed calculator, calling the objective function, logging results, and exporting a map image if desired.

- `TowerOptimizer.cpp` provides a common set of methods to evaluate f and its gradient. It is compatible with both Ensmallen and BayesOpt.

- `kernel.cu` contains all the project's CUDA code, used to move DSM data tiles to the GPU and run the new parallel viewshed algorithm on all sensors.

- `ObjectiveFunction.cpp` inputs sensor locations and calculates f by first calculating v_e for all edges and then finding the min cut by creating the dual graph.

- `elevation_map.cpp` is an optional class that outputs images of individual optimization iterations. Many of these images were used in this document, showing elevation as a background, P_c for all cells, locations and range rings for all sensors, and the min cut.

7.3 User Guide

There are many options for setting up the optimization problem and choosing an optimizer. The root folder of the code is called `CUDA Repo`, which also encapsulates most of the external libraries. The folder `CUDABatchViewsheds` contains the code and settings to be customized by the user. In there is one input folder, `Terrain`, that holds DSM data in `.bt` format. The `Output` folder is used to write results from a simulation run, including images of each iteration if desired and a table containing sensor placements and f scores for each iteration in `scores.csv`.

The user can start setting up an optimization run by specifying the width and height of the terrain region to analyze in pixels in `OptimizationRunner.cpp` and `evaluateObjective.cpp`. The number of sensors is specified in `TowerOptimizer.h` and `OptimizationRunner.cpp`. For each sensor, range and height in meters are specified in `evaluateObjective.cpp`. Probability of detection constants α and β are set in `kernel.cu`, as well as the height of the object above ground h_o.

For viewshed calculation, the number of tiles in the terrain in each dimension, the number of threads per block, and the path and location offset of the DSM data are all specified in `kernel.cu`. The tile and thread information will depend on the GPU being used. In `OptimizationRunner.cpp`, the parameters for SA and BO optimizers should also be modified to fit the problem. Details on these parameters can be found in the Ensmallen and BayesOpt documentation respectively.