Computer Science Department
CPSC 097

Class of 2007 Senior Conference on Computational Geometry and GIS

Proceedings of the Conference

Order copies of this proceedings from:

Computer Science Department
Swarthmore College
500 College Avenue
Swarthmore, PA 19081
USA
Tel: +1-610-328-8272
Fax: +1-610-328-8606
adanner@cs.swarthmore.edu

Introduction

About CPSC 097: Senior Conference

This course provides honors and course majors an opportunity to delve more deeply into a particular topic in computer science, synthesizing material from previous courses. Topics have included advanced algorithms, networking, evolutionary computation, complexity, encryption and compression, and parallel processing. CPSC 097 is the usual method used to satisfy the comprehensive requirement for a computer science major.

During the 2006-2007 academic year, the Senior Conference was led by Andrew Danner in the area of Computational Geomety with Applications in GIS.

Computer Science Department

Charles Kelemen, Edward Hicks Magill Professor and Chair
Lisa Meeden, Associate Professor
Tia Newhall, Associate Professor
Richard Wicentowski, Assistant Professor
Andrew Danner, Visiting Assistant Professor

Program Committee Members

Dan Amato
Scott Blaha
Andrew Danner
Taylor Hamilton
Phil Katz
Anthony Manfredi
Shingo Murata
Mustafa Paksoy
Alexandr Pshenichkin
Matt Singleton
Stephen St. Vincent
Giovanna Thron
Bronwyn Woods

Conference Website

`http://www.cs.swarthmore.edu/~adanner/cs97/f06/`

Conference Program

Wednesday, December 13, 2006

Table of Contents

Finding Your Inner Blaha: GPS Mapping of the Swarthmore Campus

Matt Singleton and Bronwyn Woods
{msingle1,bwoods1}@cs.swarthmore.edu

Abstract

Swarthmore College is a largely unmapped, dangerous region of southeasters Pennsylvania. Or rather, we treat it as such for the purposes of this paper. We present an interactive tool for calculating the shortest path between two points on the Swarthmore campus. We develop our tool using a combination of GPS technology and knowledge of Swarthmore's buildings. We allow users to specify a *Blaha factor*, which scales the weights of indoor paths, causing them to be treated as shorter or longer than their real lengths in the shortest path calculations. In this way, users can express a preference for traveling primarily indoors or outdoors, depending on personal preference and weather conditions.

1 Introduction

People familiar with a place often have strong intuitions about the most efficient ways of traveling between locations they frequent. However, people's intuitions are sometimes in disagreement. Additionally, special circumstances such as extraordinarily nice or foul weather may influence a person's preference for possible routes. We present a tool for identifying the shortest path between two points on the Swarthmore College campus, allowing for preferences for indoor or outdoor paths.

We mapped the outdoor paths on the campus using GPS technology and estimated the indoor paths based on our knowledge of the buildings. Using a combination of manual and algorithmic techniques, we transformed our raw point data into a graph on which we perform shortest path routing using Dijkstra's algorithm. We allow indoor and outdoor paths to be weighted differently, effectively discounting or penalizing the distance traveled inside.

Our tool is presented as an interactive GUI that allows the user to select points on a map of Swarthmore's campus. The tool graphically displays the shortest path according to the value of the *Blaha factor*, or weighting of the indoor paths, set by the user.

2 Related Work

2.1 Global Positioning System

The NAVSTAR Global Positioning System (GPS) provides precise information about location by using signals transmitted by 24 satellites in Earth's orbit. Originally designed for exclusive military use, the system was opened for civilian use as it became fully operational in the early 1990s. GPS satellites transmit ranging signals which encode information about the satellite's location at the time the signal was sent. By combining information received from several satellites, this signal allows GPS receivers to calculate their 3D location on Earth's surface. The accuracy of locations determined by GPS can range dramatically depending on the quality of the GPS receiver. Commercial quality GPS receiver units have typical errors of between 10m to 30m, while more expensive systems can reach an accuracy at the subcentimeter level (US , 2003).

Many factors contribute to the overall accuracy of measurements taken using GPS. These include, in addition to the quality of the receiver, atmospheric conditions, the environment of the user and the position of the GPS satellites relative to the user. GPS measurements with commercial receivers can only be performed outdoors and can be disrupted by dense tree cover or other large obstacles (US , 2003).

2.2 Dijkstra's Algorithm

Dijkstra (1959) describes two algorithms for finding shortest paths on a graph, one for finding the minimum spanning tree and the other for finding the shortest path between two nodes. For the purposes of this project, we were concerned only with the latter. The operation of this algorithm is discussed in section 3.3.

3 Methods

3.1 Data Collection

We collected raw data about the paths on the Swarthmore College campus using a Garmin GPSMap70 GPS unit. We marked paths by recording points manually at regular intervals. Manually recording points allowed us to determine the frequency with which we recorded points and to ensure that we recorded points at the intersections and endpoints of paths. Each point we recorded was given a unique ID, allowing us to keep track of which points started and ended any given path. In total, we collected 910 points. In addition to the data points delimiting the paths, we recorded individual points representing the doors into the campus buildings. The self-reported accuracy of the GPS unit averaged around 10m for all of our data collection.

We recorded our data using the UTM coordinate system. The UTM system breaks the globe into zones, or bands running north to south. A location is defined by its zone, an easting and a northing. The easting represents the distance from the edge of the zone, while the northing gives the distance from the equator.

3.2 Processing Techniques

Once we had gathered our raw data, we needed to make a number of additions and changes to prepare it for screen display and path computation.

3.2.1 Hand Cleanup (First Pass)

Our raw GPS data was surprisingly good, but it still contained a number of erroneous data points. At this point we had a preliminary GUI that allowed us to view the data as a collection of numbered points and lines. Given this view, it was relatively easy to identify the erroneous data points visually and then remove them by hand.

In addition to the erroneous data points, the raw GPS data is presented as one unbroken line. The result is long segments connecting the end of one path to the beginning of another. We divided the data into the individual paths again by visual examination of the data.

3.2.2 Line Intersection

Our next task was to find the points at which the paths intersected. Because the number of points in our data set was small, we decided to do this using a brute-force algorithm. Each path is made up of a number of straight line-segments, so we simply check each segment for intersections with every other segment. If an intersection is found, that point is added to both paths. This operation is $O(n^2)$ where n is the number of line segments.

3.2.3 Hand Cleanup (Second Pass)

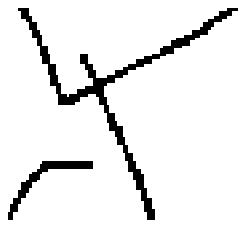


Figure 1: In the raw path data, some paths end a bit too soon, while others end a bit too late.

While GPS did a very good job of gathering data with good relative positioning (straight paths are straight and curved paths curve where they are supposed to), it did a much poorer job at absolute positioning. As a result, paths often end slightly before or slightly after they should (see figure 1). We used our preliminary GUI to visually identify where these problem areas were. We then added or removed points from the data by hand, as appropriate.

3.2.4 Adding Doors and Indoor Paths

As noted above, we gathered individual points marking the entrances to buildings in addition to our path data. Unfortunately, due to the absolute positioning problems with the GPS data, many of these points were significantly wrong. Based on our knowledge of the buildings on campus, we were able to identify which doors points were worth keeping and which we needed to be adjusted manually. GPS does not work inside, so we needed to add indoor paths by hand. We approximated these based on our knowledge of the buildings and the locations of the doors.

3.2.5 Creating a Graph

Finally, with all the paths and intersections in place, we needed to convert our data into a graph that that we could use to compute shortest paths using Dijkstra's algorithm. As our data was then, a single path could contain multiple vertices and span multiple edges. We needed to segment it so that each path corresponded to one edge in the graph, and each endpoint corresponded to a vertex in the graph. With the data in this format, it was relatively easy to build the graph structure in one pass through the data.

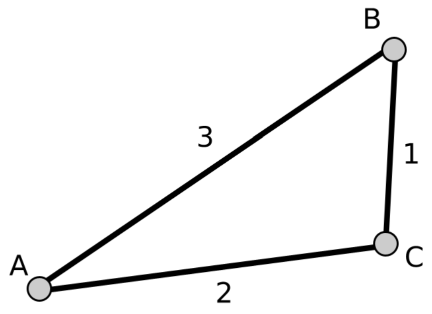


Figure 2: A simple graph.

Our graph is implemented as Python dictionary (essentially a hash table) where the keys are vertex IDs for every vertex in the graph and the value is a dictionary where the keys are vertex IDs for all connected vertices and the value is the weight of the corresponding edge. Because hash table lookups can be done in constant time, building the graph is an $O(n)$ operation where n is the number of paths.

```
{'A': { 'B': 3, 'C': 2},
  'B': { 'A': 3, 'C': 1},
  'C': { 'A': 2, 'B': 1}}
```

Figure 3: An example of our graph representation describing the graph displayed in figure 2

3.3 Computing the Path

Now that we have constructed our graph, we can compute the shortest path between any two vertices using Dijkstra's algorithm (Dijkstra, 1959). Dijkstra's algorithm, left to its own devices, will compute the entire minimum spanning tree of a graph, starting at a given node. Since all we care about is a single shortest path, we can stop computation as soon as our destination vertex is added to the tree.

The algorithm is simple, and can be easily implemented in Python. To begin, the algorithm sets the distance to the start vertex as 0 and the distance to all other vertices as ∞. Initially, the tree contains only the start vertex. The algorithm proceeds by incrementally adding adjacent vertices to the tree until every reachable vertex is added. The next vertex to be added to the tree is always the vertex whose addition will minimize the length of the longest path. Refer to figure 4 for an example.

4 Results

4.1 Raw Data

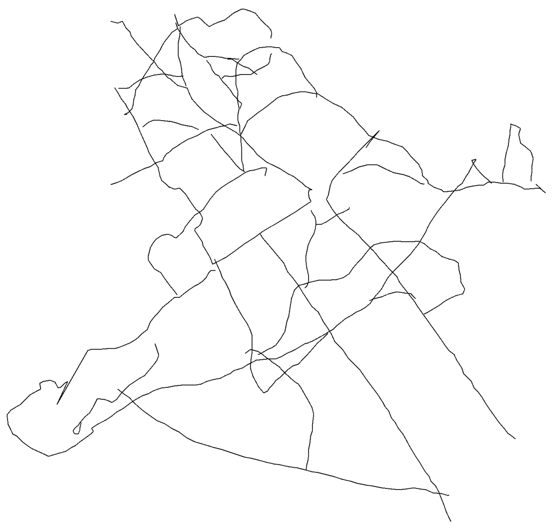
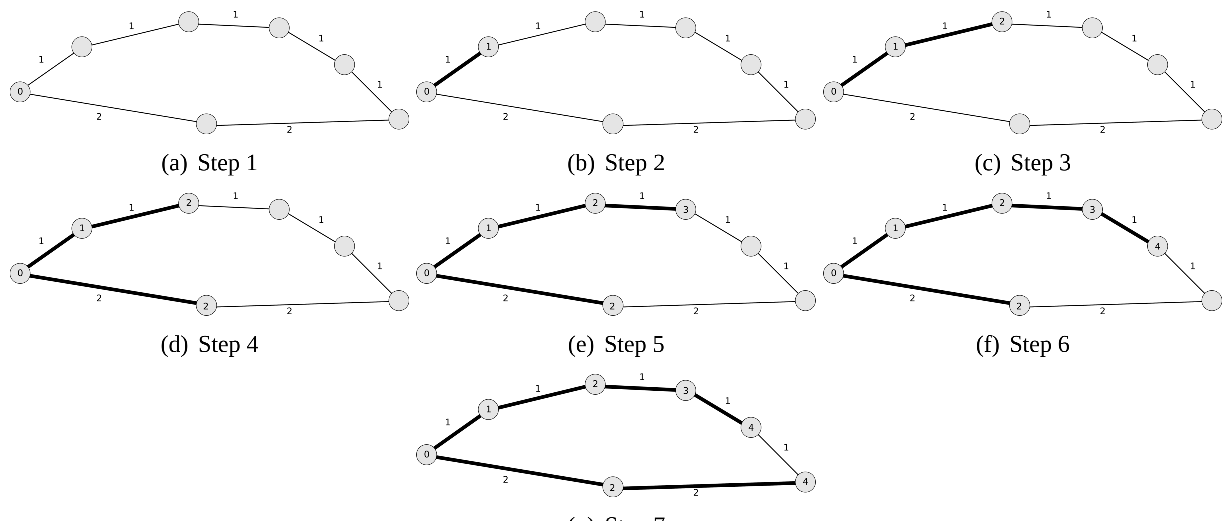


Figure 5: The raw path data from the GPS unit.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7

Figure 4: A trivial example of the execution of Dijkstra's algorithm.

Figure 5 shows our raw path data. The map is clearly recognizeable as the Swarthmore campus, but there are many flaws to be corrected. We can see several immediate problems with this data. Some paths that should intersect fall short and do not meet. Other lines do intersect, but extend past the intersection when they should end. Lastly, some points are clearly inaccurate by a large margin causing spikes in a few of the paths.

4.2 GUI

Once we had all of the underlying structure built, we created a GUI to concisely present all of the data and provide an easy method for getting user input. The GUI is comprised of three distinct areas.

Map canvas The largest and most important part of the GUI is the map canvas. This is where the map is displayed and the user can select the start and end points of their desired path. Once two points are selected, the shortest path is calculated based on the current Blaha factor and drawn own the map.

Status bar Along the bottom of the window, the status bar displays the current x- and y-coordinates of the mouse pointer as well as the current Blaha factor.

Input area To the right of the Map canvas, the input area allows the user to change the Blaha factor and reset the map.

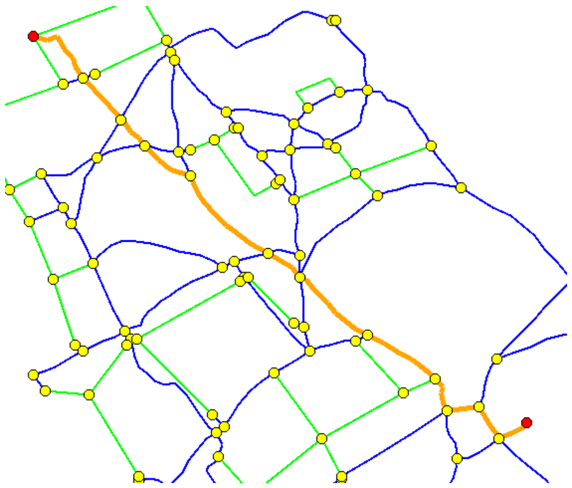
5 Discussion

Our final product is an interactive tool for shortest path routing on the Swarthmore campus. Though it might seem that the tool would be superfluous given the familiarity of students with the campus, anecdotal evidence shows that some shortest paths, especially with modified Blaha factors, are surprising even to Swarthmore students.

Though our map is created from GPS data, there are several possible sources of inaccuracy which might affect the shortest path calculations. For one thing, the lengths of the indoor paths are only estimated, and do not take into account stairs that must be climbed or doors that must be opened. We also do not consider elevation for the outdoor paths, though this is unlikely to make a significant difference.

There is a potential to expand our tool in a variety of directions. For instance, we could expand our map to cover a greater portion of Swarthmore College and the surrounding areas. We could allow the user to specify which buildings he could not pass through due, for instance, to not having a key. We might also be able to improve the accuracy of the door data points by sampling several points at the doors and averaging their positions.

Though the methods we used for creating a searchable map of the Swarthmore campus worked


(a) The shortest path with the Blaha factor set to 1.

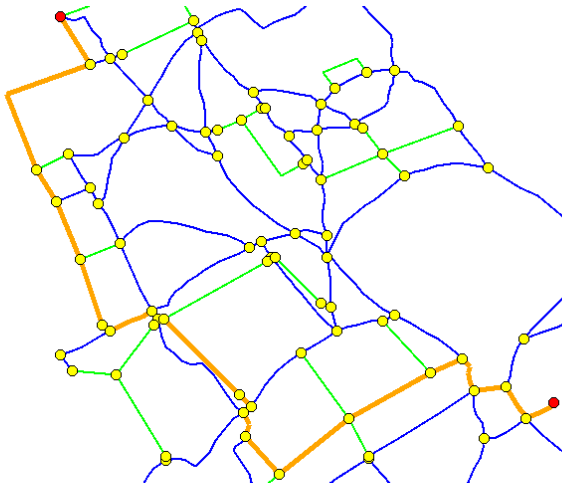

(b) The shortest path with the Blaha factor set to 0.1

Figure 6: The final display given by the GUI, showing the shortest paths from the Science Center to the McCabe Library with the Blaha factor set to 1 and 0.1.

for this task, they would not be scalable. This is due to the large amount of hand cleanup involved. However, the amount of error present in the GPS data we obtained necessitates this hand clean-up. It seems that the task of creating a map of the type we present for a larger area would require a different approach.

6 Conclusion

We present in this paper an interactive path finding tool for the Swarthmore College campus. The tool allows for differential treatment of indoor and outdoor paths, allowing the use to specify a preference for travel. Though the methods that we used for creating the map and searchable graph would not be scalable to larger maps, the techniques were effective for our task. Our interactive GUI allows the user to discover efficient paths which are occasionally surpsising even to individuals familiar with the area.

References

E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271.

US Army Corps of Engineers, 2003. *Engineering and Design - Navstar Global Positioning System Surveying*, July.

Border Patrol

Shingo Murata
Swarthmore College
Swarthmore, PA
19081
smurata1@cs.swarthmore.edu

Dan Amato
Swarthmore College
Swarthmore, PA
19081
damato1@cs.swarthmore.edu

Abstract

We implement a border patrol program that computes ideal locations for observation towers overlooking a border of interest. In this particular project, we study the border of Arizona facing Mexico. We use GRASS to manipulate elevation raster data. Our algorithm extracts the border from a raster file and locates candidate positions for observation towers along that border. The viewshed of each observation tower is computed with a direct line of sight algorithm. We employ a tower placement algorithm to select only the necessary towers from the set of all candidate towers. Our algorithm selected 92 towers within a 5km zone from the border to patrol the approximately 680km border.

1 Introduction

Border patrol is a common interest involved in national security. Militaries are frequently concerned with detecting threats along the extent of a particular border as completely and efficiently as possible. It is important that border security can be established cost-effectively as well. We model this problem as the task of placing the minimum number of towers necessary to view the entire border of interest within some range of that border.

Current GIS technology makes it possible to automate the process of planning optimal locations for border observation towers. In this paper we develop algorithms to automatically extract a border from a raster file, find candidate tower locations near the border, and select the optimal set of towers that is capable of observing the entire border.

Elevation data for many areas of interest are readily available on the Internet. We use the Geographic Resources Analysis Support System (GRASS) for data manipulation. GRASS is suited for this project because it has full functionality in visualizing elevation, data conversion, and I/O support for ascii files that are compatible with our C programs. With an abundance of digital elevation models in raster format and the software tools to manipulate them, we can successfully apply useful viewshed computation algorithms to the problem of finding the optimal set of observation towers.

The viewshed of a view point is the set of points in the terrain model that can be observed from that point. Viewshed computation is central to the automation of these algorithms because it is essential to know which border points a candidate tower is capable of observing. Our tower placement algorithm involves the iterative selection of candidate towers with the greatest contribution of yet unseen border points to the current optimal set of observation towers.

There are several parameters involved in tower placement. One of our goals is to minimize the number of towers we need to observe the entire border. This number is dependant both on the height of the towers and the maximum distance they are allowed to be placed from the border. As the height of the towers is increased, the number of necessary towers decreases, but the cost of each tower increases.

Finding a cost-minimizing solution would involve balancing these factors. Increasing the distance from the border in which towers can be placed may incorporate useful elevation maxima into the model that were previously out of range. The gains from this are limited by atmospheric conditions that limit visibility and, ultimately, by the curvature of the Earth.

We begin by reviewing the viewshed algorithm presented by David Izraelevitz in (Izraelevitz, 2003). We then describe in detail the three major steps in our project: border extraction, viewshed computation, and tower placement. Finally, we present the results in section 4.

2 Related Work

Several algorithms for computing the viewshed of a point are presented in (Izraelevitz, 2003). The first method presented is direct computation. This method essentially checks each possible obstruction on a line from the view point to the target point. If there are no obstructions along this line, then the target point is considered visible. This algorithm is straight forward, but is computationally inefficient, because it requires $O(n)$ computations for each grid point on an n x n field, resulting in an $O(n^3)$ algorithm.

The Xdraw algorithm employs the Line of Sight (LOS) function to compute the viewshed of a view point. This algorithm is faster than the direct method because it stores previous results that can be utilized at the next stage of computation along the same line.

The final algorithm improves the Xdraw algorithm by introducing a backtracking method to reduce the number of interpolations and increase the accuracy of the LOS calculations. If any point along the line between the view point and the target point coincides with a grid data point, that data point is used to initialize the LOS computation. Otherwise, the algorithm backtracks a specified distance and initializes the computation with an interpolated LOS value.

To determine border visibility, we do not need to compute the entire viewshed from a given view point. We only need to determine whether target points on the border are visible from a tower's view point. It is irrelevant to know whether the points between the observation tower and the border are visible. This negates the benefits of the Xdraw algorithm which are based on a fast incremental computation. We implement and use the direct algorithm because its inefficient computation is mitigated due to the limited number of points we are examining. In addition, its accuracy is superior to Xdraw.

Vincent presents an alternative viewshed algorithm based on ray casting in three dimensional polygonal models in (Vincent, 1999). The algorithm is based on a hierarchical partitioning of three dimensional space. This partition is represented by a k-d tree. The hierarchical partitioning of the space continues until the model of the terrain is enclosed in appropriate bounding boxes. This hierarchical structuring of the space allows Vincent to increase the search speed for view obstructions. Other optimizations include a backface culling mechanism to remove irrelevant surface polygons from the search space and a parallelization of the algorithm. Vincent also sets up an error vs. speed tradeoff by adaptively spacing the rays used to test for intersection. As the number of rays increases, accuracy improves at the cost of increased execution time.

3 Methods

Our automated border patrol algorithm has three main phases. First, the relevant border must be extracted from the data. Second, the set of candidate towers must be determined, and the viewshed of these towers must be computed. Finally, the tower placement algorithm must select the optimal set of observation towers from the entire set of candidate towers. These selections are based on the visibility characteristics of the candidate towers. These steps are elaborated below.

3.1 Border Extraction

To patrol the border, we first need the coordinates of each point on the border. It is not trivial because the digital elevation model contains no information regarding the border. Since we are looking at the Arizona border for this particular experiment, and Arizona has a straight border facing Mexico, we could have figured out the equation for the line. However, this method would severely restrict the type of border we can patrol. Therefore, we present an algorithm to extract a border in general.

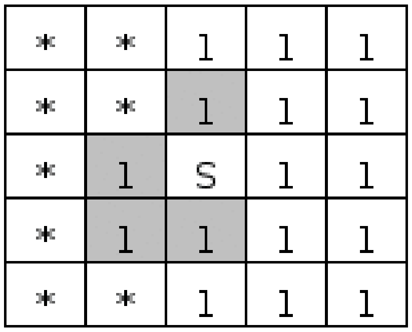


Figure 1: Start List Construction

Although we have no information about the border in the raster file, we do have a vector file that traces the border. We begin by converting the vector file into raster format using GRASS. Raster cells on the interior of the border are marked with a 1, while raster cells exterior to the border are marked with a NULL flag. We then output the raster to an ascii file, giving us a file of 1's and NULL's. Our algorithm defines a *border candidate* to be a point whose own value is 1, and also has at least one neighbor that is NULL.

Since we are working with a section of the border, we need to specify the start coordinate and the end coordinate. The algorithm checks the 8 neighbors of the start point. If any of them are border candidates, the points are added to the *start list*. It also marks all 8 neighbors as "visited". Figure 1 displays the state of the algorithm after the start list has been created. The grey cells are those that have been added to the start list.

For each point in the start list, our algorithm works as follows:

1. Dequeue a point from the start list. Call it p.

2. Look at the 8 neighbors of p, and find border candidates that have *not* been marked as "visited" yet. If there are any such points, add them to a work list. The work list is implemented as a linked-list queue, as is our temporary "border" described below.

3. Dequeue the first point from the work list and mark it as "visited." Add the point to the temporary border linked list. Call this point p and then repeat step 2. Continue until the work list becomes empty or we hit the "end" point.

4. If the work list has become empty without reaching the end, we scrap that temporary border. If we hit the end point, we have detected a potential border. In either case, we restart the process from step 1 with another start point, until we exhaust all the start point possibilities.

Once all start points have been expanded, we may have two valid borders: one for each of the two directions along the border that lead from the start point to the end point. We must determine which of the two potential borders is the real border of interest, and which is simply the remaining border of the region which excludes the border of interest. For now, we make this decision by simply choosing the shorter border.

We then copy the linked list into an array, as arrays are easier to work with for our purpose. At this point, each border cell all has height of 1 since they were extracted from the 1 or NULL raster. It is important that we go through the actual elevation raster and set the elevation of these border points to their true values.

At this point we have extracted the border between the start point and the end point. We also define the *zone* in which towers may be placed. This zone is a band of specified width along the extracted border and inside of the "home" territory. To find the points falling in this zone, we essentially perform a breadth first search of the 1 or NULL raster. We begin with the extracted border points, and add them into a queue. While the queue is not empty and the specified width has not been reached, we dequeue a point from the queue. This point is flagged as "visited," and all of its unvisited, home territory neighbors are added to the queue. All of the points in the zone are stored in an array for future use.

3.2 Viewshed Computation

The viewshed of a view point in a terrain model is the set of all points visible from that point. To patrol the border we do not need to compute the full viewshed of a tower location. We only need to compute the visibility of points on the border. If a set of towers can collectively view each border point, then the border is considered to be fully patrolled by that set.

As we are not interested in computing the full viewshed of potential tower points, but only the visibility of the border points, we do not need the optimized approximation algorithms presented in (Izraelevitz, 2003). We can afford to simply compute the visibility from the potential tower point to each target border point. The relevant points in the viewshed computation are the view point, p_v, the target point, p_t, and the point of potential view obstruction, p. The visibility of p_t from p_v can be determined with equation 1.

$$elv(p_t) > elv(p_v) + \frac{|p_t - p_v|}{|p - p_v|}(S(p) - elv(p_v)) \quad (1)$$

$elv(p_a)$ denotes the elevation of point p_a. $|p_a - p_b|$ represents the Euclidean distance between two points, p_a and p_b. Finally, $S(p_a)$ is the interpolated elevation of the terrain model at the (x, y) coordinates of p_a.

This inequality places a constraint on the height of the target point, p_t, based on the elevation data along a sight line between the view point and the target point. The inequality generates the minimum elevation at the target location that is visible from the view point given the elevation of the potential obstruction at point p. If the actual target elevation is less than or equal to this minimum, then the target point is not visible from the view point because the sight line is obstructed by the obstacle at point p.

To determine the visibility of the target point, equation 1 must be evaluated at each point, p, along the sight line between the view point and the target point. If no point along the sight line obstructs the view, then the target is visible. We approximate this test by evaluating equation 1 at regular intervals along the sight line between the view point and the target point. The theoretical sight line will fall between two grid data points in general. The elevation at point p is the interpolated elevation of the two grid data points on either side of the sight line.

The interpolated elevation at point p and the elevation of the target point are adjusted for the Earth's curvature prior to the computation of equation 1. The curvature adjustment of a point p_a's elevation, $\Delta_{elv(p_a)}$, is given by equation 2, where d represents the quantity $|p_a - p_v|$. The geometry of the approximation is illustrated in figure 2.

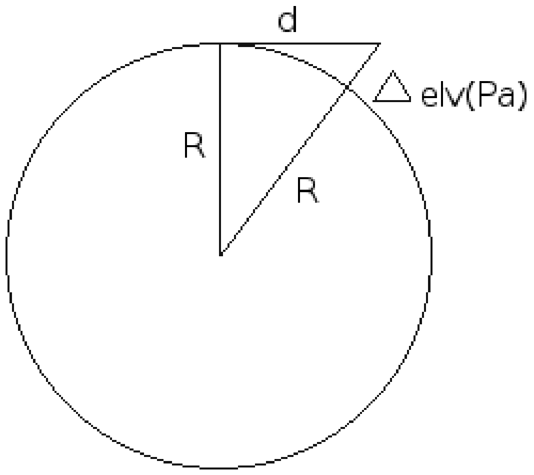


Figure 2: Geometry of the Earth Curvature Approximation

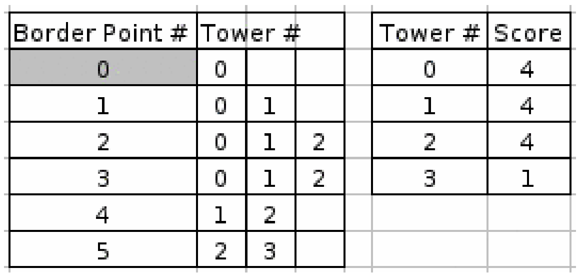
$$\Delta_{elv(p_a)} = \sqrt{{R_{Earth}}^2 + d^2} - R_{Earth} \quad (2)$$

We use this viewshed computation method to compute the visibility of each border point from a potential tower location.

3.3 Placement Algorithm

Our goal is to view the entire border with as few towers as possible. As noted in section 3.1, there is a zone of specified width back from the border in which towers may be placed. We base our tower placement on the principle that an observer can see more if s/he is at a higher elevation. We begin by locating all of the local maxima with in the zone. A local maximum is defined as a raster cell having no higher neighbors within the zone. These maxima serve as the candidate positions of our towers.

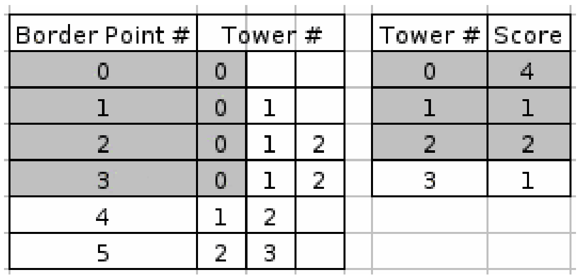
We first figure out which candidate towers are absolutely necessary. A candidate is considered necessary if there are points on the border that can only by seen by that candidate position. Therefore, instead of computing which points each tower would be able to see, we compute for each point on the border, which towers can see it (although they are computationally equivalent).



Border Point #	Tower #		
0	0		
1	0	1	
2	0	1	2
3	0	1	2
4	1	2	
5	2	3	

Tower #	Score
0	4
1	4
2	4
3	1

Figure 3: Example of border visibility state



Border Point #	Tower #		
0	0		
1	0	1	
2	0	1	2
3	0	1	2
4	1	2	
5	2	3	

Tower #	Score
0	4
1	1
2	2
3	1

Figure 4: Example of updated border visibility state

We maintain a table of integers, with each row corresponding to a point on the border. In each column, we keep track of a tower identifier that can see the point. For each point on the border, we test whether that point can be seen by each of the candidate positions. If the point is visible from a candidate tower location, we add the identifier for that tower to the point's row in the table.

Our table may look like figure 3, which shows that the first point on the border can be seen by Tower 0, the second by Tower 0 and 1, the third by Tower 0, 1 and 2, and so on. Every time a tower is added to the table, we increment the "score" of that candidate tower by 1. The score corresponds to the number of points the candidate tower could see if it were built.

After we have created the table, we traverse the table to find the border points visible from only one tower. We know that the single towers that can see these points are critical. In the table above, for example, Tower 0 is critical because it is the only tower that can see the first point of the border.

Every time we find a critical tower, we add it to the permanent tower list. Then we traverse the integer table to "close" all border points that can be seen by that tower. We traverse the list, find out which points that tower can see, and, flag that point as "secured." We also traverse the row for the secured point, and if any other towers can see it, we *decrement* the score of those towers. By decrementing the scores of these towers, we ensure that the score represents only the number of unsecured points viewable from the candidate tower. This prevents redundant towers which can view few new border points from being selected as good candidates in future iterations of the algorithm. The effect of adding Tower 0 to the permanent list in our example above is illustrated in figure 4. The score of Tower 1 is decremented by three because three of the border points it can view are made redundant with the addition of Tower 0 to the permanent list.

Every time we add a tower to the list, we check to see if the entire border has been "secured" yet. The chances are, the border cannot be secured by just those critical towers. So we move on to the second phase of the placement algorithm, in which we simply pick the candidate tower with the highest score out of the remaining towers (when we put a tower into the permanent tower list, we set its score to -1, so we never look at it again). When we add the tower, we "close" the points it can see as before. We repeat this process until the entire border is secured, or there are no more candidate towers to consider.

4 Results

The input raster, which covers the southern half of Arizona, has 2846 x 5705 data points at 100m resolution. When we extract the border, we find that it has 6768 data points. Out of the 6768 points, 1332 are local maxima, so we begin with 1332 candidate points for the towers.

We begin by restricting tower placement to points directly on the border. When we scan for points visible from only one tower, we find that only 6 towers are critical. With those 6 critical towers, we can view 844 / 6768 border points. The next tower we place, the tower with the highest score, can view 619 points by itself. After securing 6642 / 6769 points, our additional towers can only see 5 novel points. We also need 15 towers that can only see 1 unique point. In total, we need 120 towers.

When we expand the zone, we have many more points to work with. Within a 5km zone from the border, we have more than 370,000 points, from which we obtain 8974 maxima. While we have

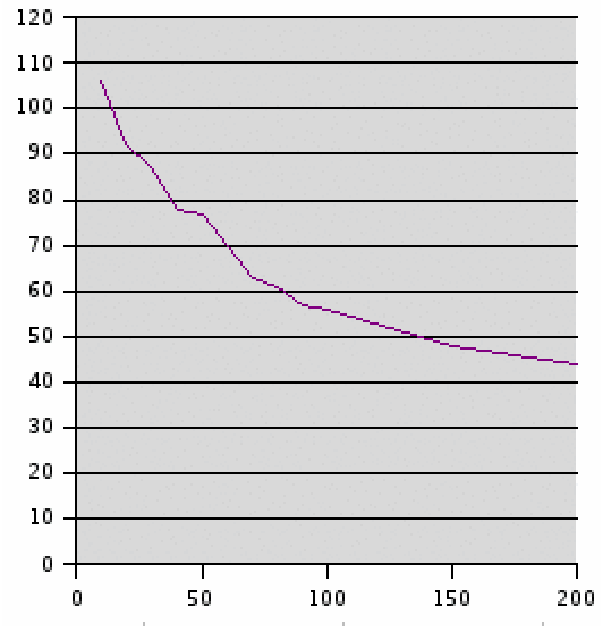


Figure 7: Number of Towers vs. Height of Towers (meters)

about 50 times as many total data points, we only have about 7 times as many maxima because we are now checking all 8 neighbors to test if a point is a maximum, as opposed to just checking the 2 adjacent points on the border. For 20m towers, by expanding the border zone to 5km wide we decreased the number of necessary towers to 92.

5 Discussion

Considering that the border is approximately 680km long, the number of towers we need is relatively low. With 20m towers in a 5km border zone, we need 92 towers. On average, each tower is responsible for roughly 70 data points, or 7km of the border. The number of towers seems to be higher than it could be due to the clusters that we can observe in figure 6. The clusters suggest that our algorithm may not be suited to certain geographical features found in these regions.

Still, we can infer some useful information from the data. We expected the number of towers required to decrease as we increased the tower height, but as we can see from figure 7, it seems to be asymptotic. In fact, when we tested with kilometer-high towers (which is absurdly high), we found that we would still need 23 towers. What it tells us is that after

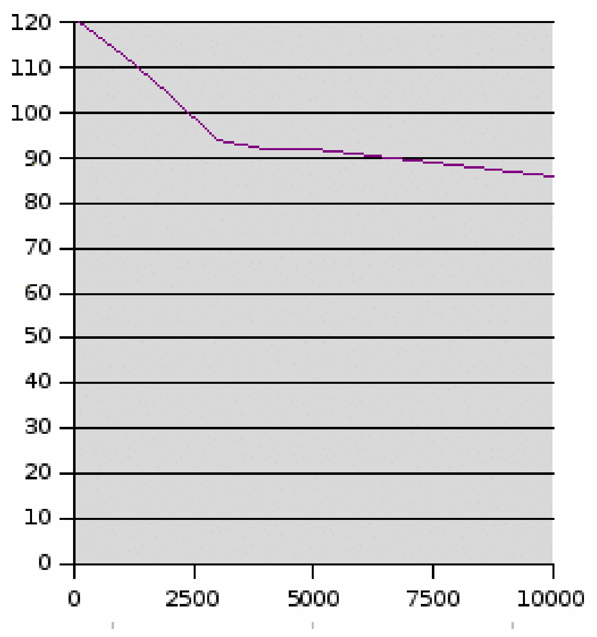
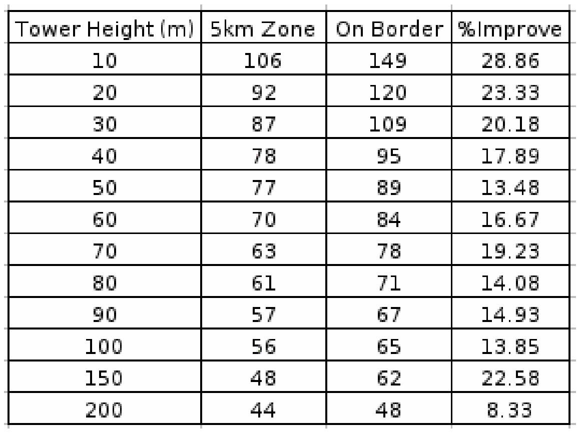


Figure 8: Number of 20 meter Towers vs. Width of Border Zone (meters)



Tower Height (m)	5km Zone	On Border	%Improve
10	106	149	28.86
20	92	120	23.33
30	87	109	20.18
40	78	95	17.89
50	77	89	13.48
60	70	84	16.67
70	63	78	19.23
80	61	71	14.08
90	57	67	14.93
100	56	65	13.85
150	48	62	22.58
200	44	48	8.33

Figure 9: Effect of Zone Size on Number of Necessary Towers

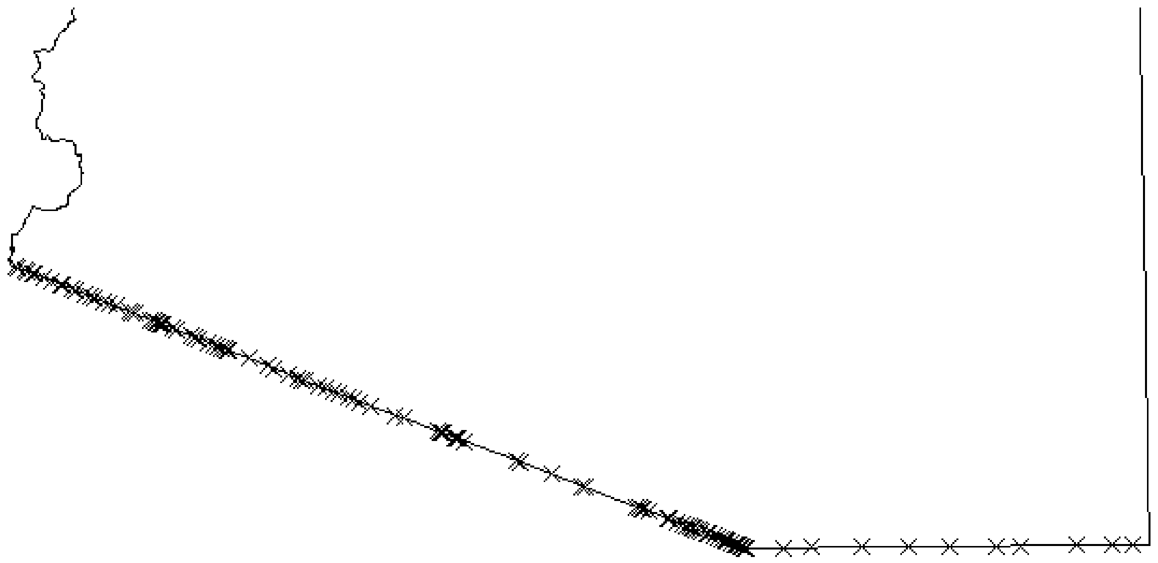


Figure 5: Tower Locations Directly On The Border

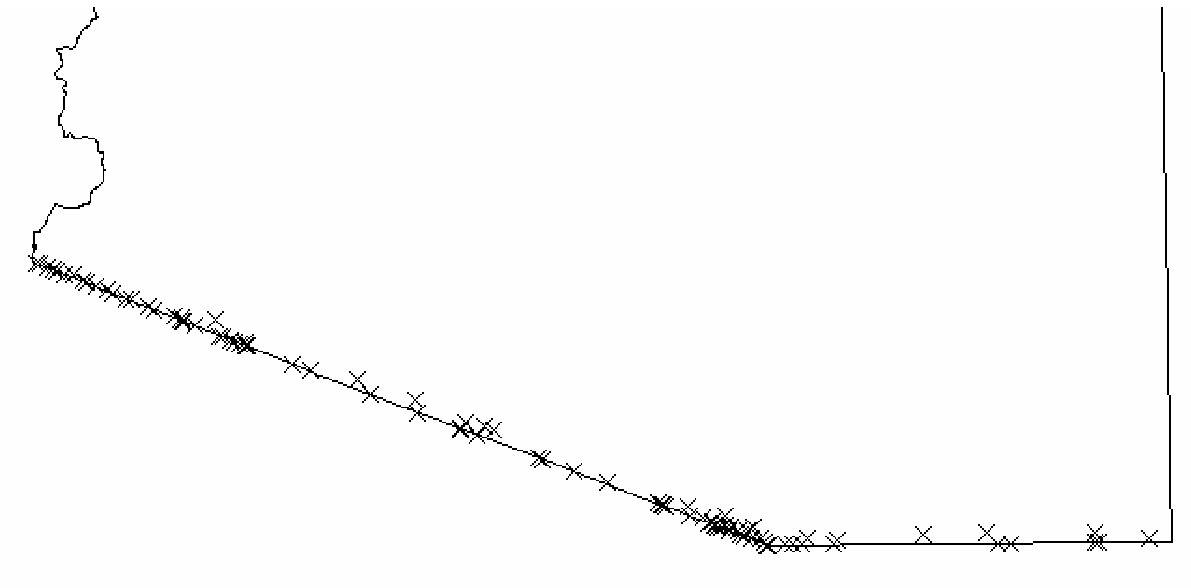


Figure 6: Tower Locations Within A 5km Zone

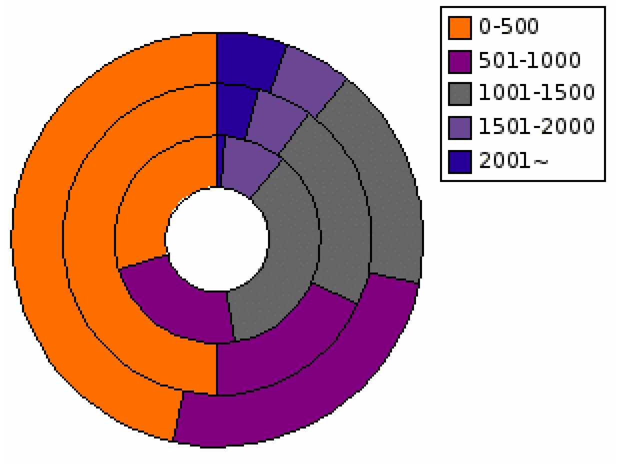


Figure 10: Pie Chart Overlay of Zone Point Elevation, Maxima Elevation, and Tower Elevation

a certain point, we do not gain as much from making the towers taller. By comparing our data with the cost of building towers of varying height, we can theoretically obtain the optimal tower height in terms of cost.

We can also see that expanding the zone from just on the border to some distance away from the border tends to improve performance. As seen in figure 9, we can reduce the number of towers (ranging in height from 10 meters to 100 meters) by between 13.85% and 28.86% by allowing a 5km zone. The number is bound to be inconsistent, because it depends on how many more important maxima we can gain by backing up, and the definition of an important maxima depends on the tower height. We also note that this behavior also seems to be asymptotic, even more so than the tower height. By moving away from the border, we are expanding the visible region, but we are also making ourselves susceptible to more obstacles on the way. Also, we may lose some visible points due to the Earth's curvature. For 20m towers, it looks like 3km away from the border is the best distance. Still, it seems to be very important that we actually use the zone to improve our performance rather than rely on increasing the tower height. For 20m towers, the 23% improvement due to the 5km zone is roughly equivalent to improvement that could be gained by increasing the tower height to 50m.

The distribution graph in figure 10 shows the elevation distribution of 3 categories: the outermost ring corresponds to the entire border zone, the middle ring is the local maxima, and the inner ring shows the actually placed towers. Local maxima are distributed fairly evenly across the zone, as inferred from the graph. In contrast, we can tell that the algorithm preferred to place the towers on higher points. Even though majority of the region is low elevation (0 - 1000m), almost half of the towers are placed at elevation 1000m or higher. We had few towers placed in elevation 2km or higher relative to the region, but it is mostly because all the high elevation points are clustered near the east end of the border, and if we place a few towers in the elevated region their visible field should be large, negating the benefit of more towers in that region.

A major problem with our algorithm is its computational complexity. Disregarding the complexity of file I/O, it first takes $O(n)$ time for locating the maxima, where n is the length of the border. Then scanning the points on the border costs $O(n^3)$ time (each border point $\times$ each candidate point $\times$ linear viewshed computation, though we expect the linear viewshed computation not to be as large as n). The addition of critical towers to the permanent tower list requires another $O(n^3)$ time, as we need to traverse the table to find each point visible from the critical tower. For each point found, we need to traverse its row in the table again to "close" the point and decrement the score of all other towers that can see the point. Again, the last step usually is much smaller than n, as we do not expect all towers to be able to see the same border point. When placing 20m towers on top of the border, our running time was around 90 seconds after extracting the border. As we increase the border zone, running time increases, as is the case when we increase the tower height. This happens due to the method of our viewshed computation. Because we are doing linear line of sight from a point to a point, as soon as we see an obstacle high enough to block the target point on the direct line, we can stop the computation. When more towers tend to be visible from the border, the number of computations required increases.

6 Conclusion

We have determined that the border of Arizona can feasibly be patrolled by observation towers of reasonable height and reasonable distance from the border. We expect that this algorithm could be successfully applied to any border. The execution time of the algorithm for a large data set was not prohibitive, and we conclude that the direct line of sight method for determining visibility is sufficient for the border patrol problem. Finally, we conclude that there are diminishing returns in terms of visibility as tower height and distance from the border are increased. Given appropriate cost parameters, the most efficient means of patrolling the border could be obtained with our method.

7 Acknowledgements

We would like to heartily thank seamless.usgs.gov for presenting us with the elevation data.

References

David Izraelevitz. 2003. A Fast Algorithm for Approximate Viewshed Computation. *American Society for Photogrammetry and Remote Sensing*, 69.7.

Andrew Vincent. 1999. Terrain Occlusion Using Binary Adaptive Ray Casting. *Silicon Graphics Inc.*

Parallelized Interpolation: A Quantitative Assessment

Scott Blaha
Swarthmore College

Mustafa Paksoy
Swarthmore College

Abstract

The conversion of raw point-cloud elevation data to grid DEMs is done by interpolation. Current interpolation methods produce either high quality results or take an acceptable amount of time, but not both. This paper seeks to reconcile these two objectives through parallelizing a high-quality interpolation method, nearest-neighbor averaging. We explore the speed-up obtained by parallelization and compare run-time with the lower quality binning method.

1 Introduction

LIDAR, one of the primary forms of elevation data collection, yields a cloud of elevation data points. However, Geographic Information Systems (GIS) like GRASS often require data in the form of a digital elevation model (DEM). One of the simplest methods to perform this conversion is called *binning*. Binning simply averages the points in each grid cell of the DEM to yield an elevation for the grid cell. However, because of the non-uniform nature of the point cloud, some of the grid cells of the DEM might not contain any points. Thus, it is possible to have holes in the DEM after binning.

The solution to this is to use one of a set of methods known as *interpolation*. A typical simple linear interpolation might take an average of points close to an empty cell, weighted by distance from the cell. Unfortunately, if there are n points, then there are potentially $O(n^2)$ interpolation calculations to be performed. This fact, paired with the typically extremely large data sets of interest in GIS, makes interpolation a highly non-trivial task. In fact, in a recent I/O-efficient point cloud to DEM algorithm (0), from 52% to 86% of running time was spent interpolating depending on the data set. Clearly, a faster method of interpolation is needed.

The basic trade-off in interpolation is quality (e.g. representativeness) of the resulting DEM versus the computational complexity of the interpolation. Rather than deal with reduced DEM quality in our quest for better interpolation run-times, we will parallelize the interpolation of point cloud elevation data. Because of the locality of reference of the interpolation task, parallelization can provide an exponential reduction of the time to interpolate a set of points, based on the number of computers.

2 Methods

2.1 Serial Binning

Our first method is a simple implementation of serial binning. A grid cell's value is the average of all points from the point cloud in that cell. A grid cell containing no points is assigned a “no value” constant; in our case, this was -99999. Serial binning will be our base-line for comparison of competing interpolation methods, both in terms of interpolation quality and run-time efficiency. We hope that parallelization will speed up creating a DEM, and that interpolation will improve the quality of the DEM. Since it takes constant time to place a point into a bin, the run time complexity of binning is $O(n)$, where n is the number of input points. However, the constant hidden in this notation was experimentally shown to be quite small.

2.2 Parallelized Binning

Next, we implemented a parallelized binner. This simply splits the task of binning and averaging points between several computers. It will produce the same interpolation as a serial binner, but hopefully with a slight efficiency boost. We do not expect to see much improvement by using this method.

Our parallel applications use the Message Passing Interface (MPI) standard for distributing and collecting data. MPI provides various facilities for passing data around and synchronizing computation. In our case, we just needed to send data back and forth. We use the Local Area Multi-computer (LAM) implementation of MPI, it lets us set up virtual clusters using an arbitrary number of nodes in the Computer Science Department network. These machines are all on the same subnet, so we expect network bandwith to be high and latencies to be low.

Initially, we split data into equal size y-intervals. This led to different hosts creating overlapping grids, which greatly complicates the process of merging results. To ameliorate this, we increase the y-interval so it becomes a multiple of the bin height of the grid (see Figure 1). So, the splits in the data align with the grid, which prevents different hosts from creating overlapping grids.

2.3 Nearest-Neighbor Interpolation

We have implemented a simple type of interpolation, *nearest-neighbor* interpolation. In this method, we first calculate the *centroid* of each grid cell, the center point of the cell. Then, we sort points by their Euclidean distance from the centroid. Finally, we set the elevation of the grid cell to be the average of the k points nearest to the centroid. Experimentation showed that 10 is an acceptable value for k.

Because sorting is $O(n \log n)$, if we have g grid cells and n points, then this algorithm has a run-time complexity of $O(g \cdot n \log n)$. This is because we must sort all the points based on distance from each cell's centroid. As we note below, this method is unacceptably slow. However, it does not suffer from the "holes" in the grid that binning does.

2.4 Parallelized Nearest-Neighbor Interpolation

Parallelization of our interpolation algorithm proceeds in much the same way as with simple binning: we break the grid into a number of approximately equal sections, one for each host, and send each host only the points which fall in that bin. Each host then performs the nearest-neighbor interpolation described above.

The theoretical speed-up we should see is more than quadratic in the number of hosts. If we have h hosts, and the points are approximately evenly distributed over the grid, then each host will get about $\frac{g}{h}$ of the grid and $\frac{n}{h}$ of the points. So, each host will have a runtime complexity of $O(\frac{g}{h} \cdot \frac{n}{h} \log \frac{n}{h}) = O(\frac{gn}{h^2} \log \frac{n}{h})$. Our tests have supported this analysis: we do in fact get super-quadratic speedups from adding hosts (see Section 3).

2.5 Smoothing Parallelized Interpolation

Unfortunately, parallelization can result in edge effects in interpolation results. Along the edge of a sub-grid sent to a host for interpolation, the closest elevation data points to a centroid might be in a different sub-grid, and thus are not considered during the interpolation. This can result in noticeable lines or bands in the resultant DEM. We have called our solution to this *smoothing*. We pass the points belonging to the immediately surrounding sub-grids along with the points in the sub-grid we send to each host. This approximately triples the number of points sent to each host, but results in less noticeable edge effects.

3 Results and Discussion

We have fully implemented the algorithms described above. We ran tests on a 100,000 point subset of a LIDAR-generated 100 foot resolution point cloud. This subset was picked by sorting the 2,000,000 points by y-value, and picking every twentieth. See Figure 2 for visualizations of the interpolated data set. Interpolation is the clear winner in the quality department - there are no holes, and the resultant DEM looks like a "filled-in" version of the binned DEM (Figure 2(a)). It is hard to notice the difference between smoothed and non-smoothed DEMs with the naked eye, but Figure 2(d) shows the result

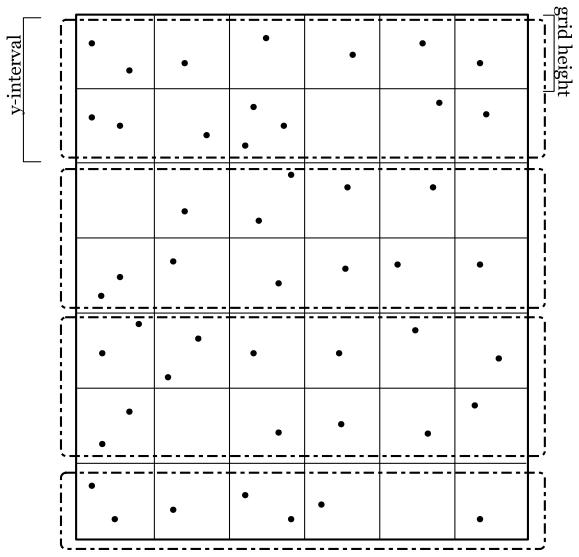


Figure 1: Splitting the grid into sub-grids for parallelization

of subtracting the two DEMs from each other. Notice that the difference lies along the cuts between the sub-grids sent to different hosts. This shows that smoothing actually does what it is supposed to, that is, it smooths out the edge effects between sub-grids.

The actual running time of each algorithm was calculated several times to test the possible speed-ups obtainable by parallelization. Unexpectedly, parallel binning is actually slower than serial binning (see Figure 3). We believe this in due to the overhead of passing data over the network compared with the blazing speed of the simple binning algorithm. In the case of interpolation, we observed the extreme parallelization speed-ups predicted above. See Figure 4 for a chart of running time versus number of hosts - note that the time axis is logarithmic. So the predicted super-quadratic speed-up does occur. As a simple comparison between binning and interpolation, with 100,000 points, 20 hosts, and a 100 foot resolution, binning takes 1.4 seconds, non-smoothed interpolation takes 24.8 seconds, and smoothed interpolation takes 73.3 seconds. Refer to Figure 5 for a comparison between the three methods

4 Conclusion

Parallelization provides an excellent speed-up for interpolation methods, but does not decrease the running time of binning. Before parallelization, our interpolation method was intolerably slow, but with 20 hosts, its runtime is very reasonable. Smoothing is also a desirable option when using parallelized nearest-neighbor interpolation, however it approximately triples the run time of the interpolation. For casual use, parallelized non-smoothing interpolation suffices. However, if more accuracy is required, then smoothing provides that with only a three-fold slowdown.

Future work in this area could include parallelizing the extremely popular quad-tree interpolation algorithm. Also, we can optimize the nearest neighbor interpolation by using a scan-line approach and only sorting a section of the whole data set a time.

References

P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In Andreas Riedl, Wolfgang Kainz, and Gregory Elmes, editors, *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling*, pages 771–788. Springer-Verlag, 2006.

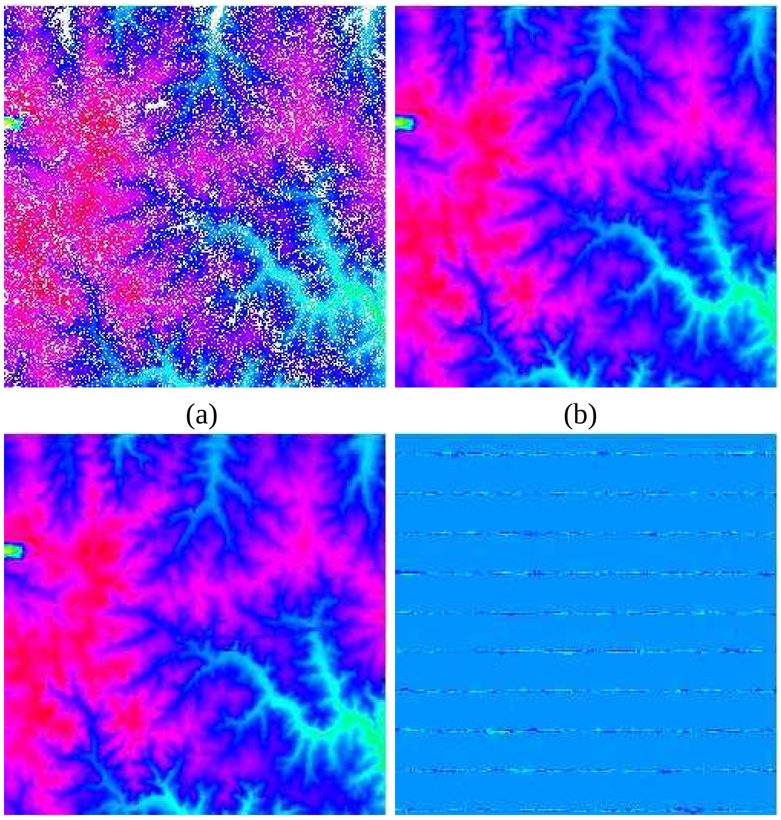

(a) (b) (c) (d)

Figure 2: Results of interpolating sparse data at 100 feet. (a) is the result of binning - note the cells with no value. (b) is our parallel interpolation algorithm without smoothing, and (c) is with smoothing. (d) is the difference between (b) and (c).

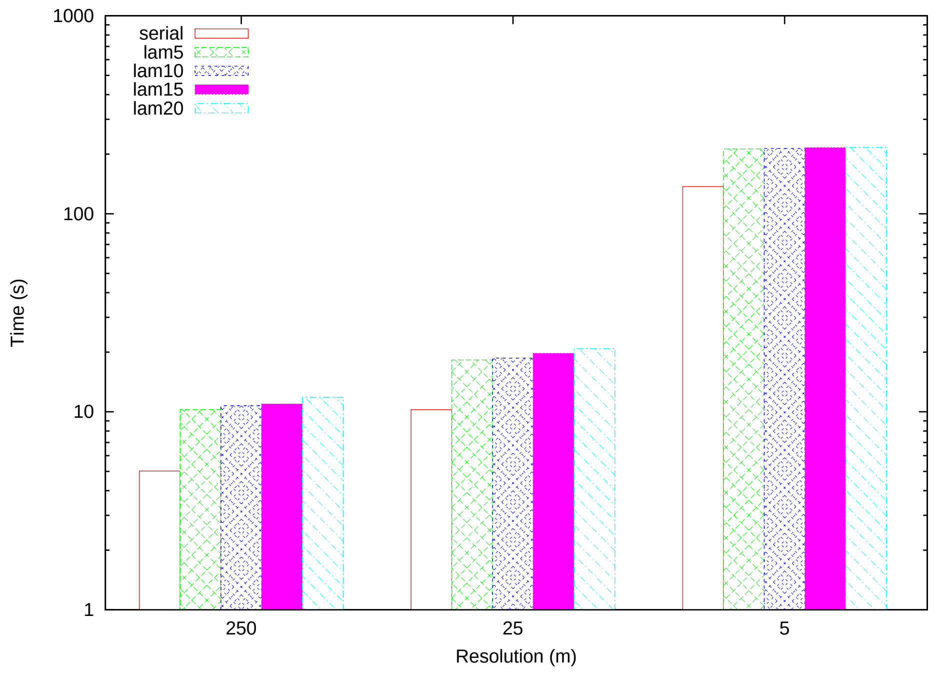


Figure 3: Comparison of serial and parallel binning running times.

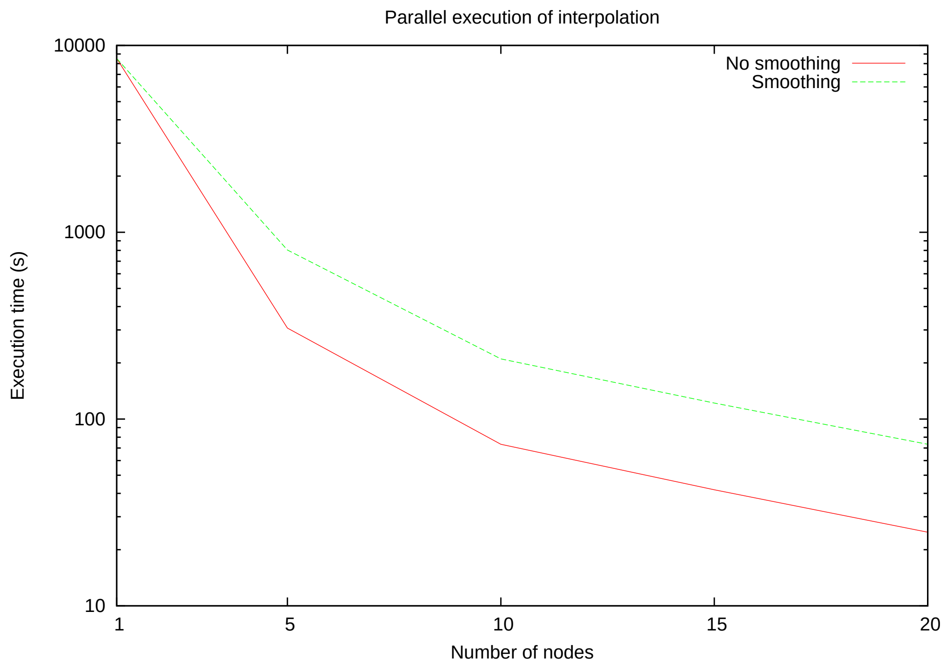


Figure 4: Comparison of non-smoothed and smoothed interpolation running times, showing super-quadratic speed-up for adding hosts.

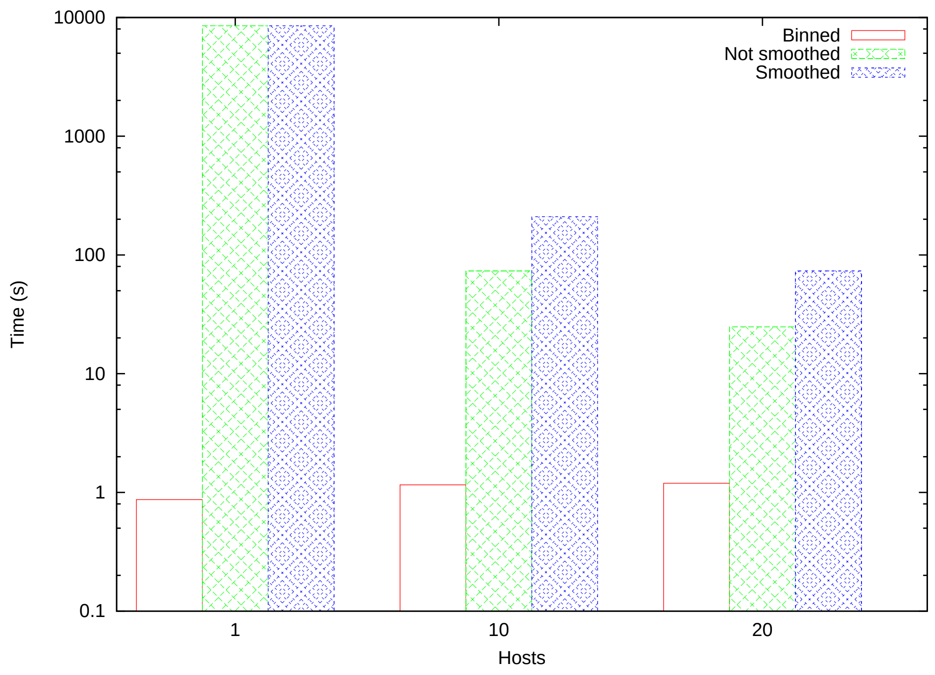


Figure 5: Comparison of parallel binning, non-smoothed interpolation, and smoothed interpolation running times.

Bridge Detection from Elevation Data Using a Classifier Cascade

Anthony Manfredi
amanfred1@swarthmore.edu

Alexandr Pshenichkin
apsheni1@cs.swarthmore.edu

Abstract

Bridges and similar non-obstructing features inhibit correct flow routing on high-resolution digital elevation models because their apparent elevation does not reflect the elevation at which water may pass underneath them. Our goal is to identify such features using the elevation data so that flow-routing algorithms may find paths under them correctly. We use an algorithm based on Viola and Jones' object-recognition system. Simple filters are applied in sequence to efficiently narrow the search space down to a final set of likely candidate features. This paper presents a successful system for identifying bridges that can be fairly easily integrated into existing GIS systems.

1 Introduction

New hi-res terrain scanning techniques such as laser altimetry (lidar) have greatly expanded the accuracy of GIS. The improved resolution has introduced many new details into digital elevation maps; many such features, however, hinder analysis of the underlying bare-earth terrain. One of the most important problem features are bridges. From the air, a bridge appears as a solid ridge, but, in reality, water can pass beneath it. While a raw data dump may contain some points that are visible underneath a bridge, current preprocessing techniques will tend to remove these, leaving a solid obstacle on the processed digital elevation model (DEM). This confuses flow-routing algorithms, which must flood terrain or search for convoluted detours to escape the local minimum created by the presence of the false ridge. Our goal is to identify bridges and similar features, such as drainage tunnels, on digital elevation models, so that water flow can be routed through them. Appropriate flow routing can be accomplished with minimal modification of existing algorithms by simply cutting through a bridge once it is marked out.

1.1 Related Work

Sithole and Vosselman (Sithole and Vosselman, 2006) describe a system for the geometric recognition of bridges as part of a general system for creating bare-earth data from raw lidar input. Their system looks for features that drop off sharply on two sides and fade smoothly into the surrounding terrain on the others. Calculating and analyzing bounding polygons for terrain features, however, is computationally intensive.

Our algorithm is inspired by computer vision research by Viola and Jones (Viola and Jones, 2002). Their system utilizes a "cascade" of simple filters, each of which is sensitive to a specific pattern. The algorithm reliably recognizes faces in real-time video. They also suggest a technique for fast computation of rectangle sums, called the integral image method. Each pixel in the integral image is the sum of the values of the pixels above and to the left of its location in the original image, which allows any rectangle sum to be computed with only four addition operations if the integral image already exists. This technique allows us to quickly calculate statistics for subsections of the map. For example, finding the average elevation in a ten-by-ten square area conventionally would require adding together one hundred values; with the integral image method, we need only access four values (the corners of the box) to get the area sum.

2 Methods

2.1 Algorithm

Our system is an implementation of the cascade concept of Viola and Jones in a novel domain. A sliding window moves over the map, examining small sections of the terrain in sequence. The window may move one or several pixels at a time: this is the step size of the window. A larger step size decreases runtime significantly but also decreases accuracy. Empirically we determined that a step size of 2 pixels did not result in a significant decrease in accuracy.

Each window is passed through a series of filters. A filter is a function that evaluates the pixels within the window statistically or geometrically and decides to accept or reject the slice. To save storage space, the algorithm applies all the filters to each window in order before moving on to the next; this way, no intermediate candidate lists (which could be quite large) are stored in memory. If any filter rejects the slice, it ceases to be relevant and the window moves to the next target. Like in the Viola-Jones algorithm, the collective action of the filters makes up for their individual inaccuracy. It is important for each individual filter to have a very low rate of false negatives, so that they do not reject good candidates prematurely.

In order to accommodate bridges of varying sizes, we make several passes over the map, changing the scale of the window each time. One can reasonably expect bridges to be at least one car lane and no more than a dozen lanes wide, and filters must take in some of the surrounding area for comparison as well. We are currently using window sizes of 100, 150, 200, 250, 300, and 400 feet in an attempt to accommodate all reasonably-sized bridges.

2.2 Filters

We have implemented several filters to detect bridge-like features. Since the overall goal is to aid hydrological modeling, we focus on discovering terrain elements that have a strong effect on existing flow-routing algorithms and trying to identify them as bridges.

1. The *high gradient* filter accepts an image if at least ten percent of the pixels in the filter window have a gradient above a certain threshold. Currently this threshold is 2.4 feet of elevation per 10 feet of translation (empirically determined), but we may adjust it in the future and analyze how it affects our results. This filter is designed to find the steep edges of bridges.

2. The *flood fill* filter accepts an image if at least thirty percent of the pixels in the filter window were flood-filled by a flow-routing algorithm. This filter is designed to capitalize on the fact that bridges in general, and particularly the bridges that we want to remove to do correct flow routing, cause flood filling along their length.

3. The *minimum fill depth* accepts an image if there is at least one pixel in the window that was flood-filled higher than 8 feet. This filter is designed to focus on areas that are significantly problematic for hydrological modeling.

4. The *low gradient* filter accepts an image if at least twenty percent of the window area is low gradient pixels, where the low gradient threshold is 0.5 feet of elevation per 10 feet of translation. This filter is designed to look for the flat area of the bridge itself.

5. The *minimum elevation difference* filter accepts an image if the difference in elevation between any two pixels in the window is above 7 feet. This filter capitalizes on the fact that bridges will be elevated above the surrounding terrain.

6. The *height bridge shape* filter accepts an image if a stripe down the middle third of the image matches a low:high:low elevation pattern when compared to the average elevation the entire window. This filter is rotated eight times at pi/8 radian intervals to catch varying bridge orientations. If any of these rotated filters match, the image is accepted. This filter is designed to find an elevation pattern that looks like a bridge: high in the middle and lower on the sides.

7. Much like the height bridge shape filter, the *gradient bridge shape* filter accepts an image if a stripe down the middle third of the image matches a high:low:high gradient pattern, using the same thresholds for low and high gradients that were used in the previous gradient filters. This filter is also rotated in the same manner as filter 6. This filter is designed to find a gradient pattern that looks like a bridge: flat in the middle and sharp on both sides.

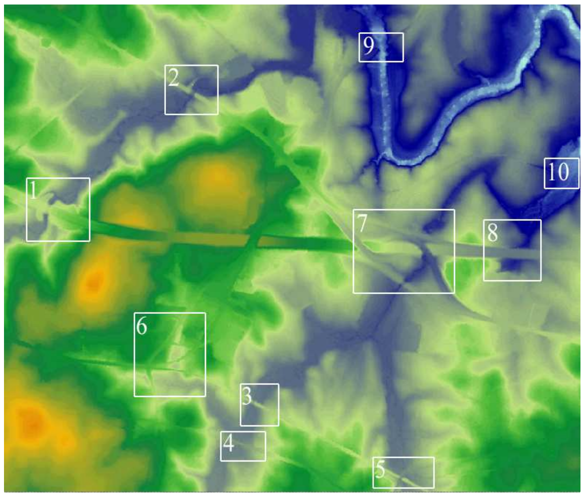


Figure 1: A DEM with hand-labeled features.

3 Results

We focused our testing on an area outside Durham, where Interstate 85 crosses highway 70. The combination of multiple roadways and a winding stream produce numerous interesting features to analyze.

Figure 1 shows a DEM of the area with hand-labeled features. Notable elements in this image are:

- Feature 1 (seen in detail in Figure 2) is a drainage pipe under a roadway. While shaped very differently from a bridge, it serves the same hydrological purpose, allowing water to pass beneath it.

- Features 2, 3, 4, and 5 are, very distinctly, bridges. Some (particularly 2 and 5) appear to have been precut. While it appears to be less pronounced than the others, Feature 4 has not been cut by the preprocessor, so it is of interest to us.

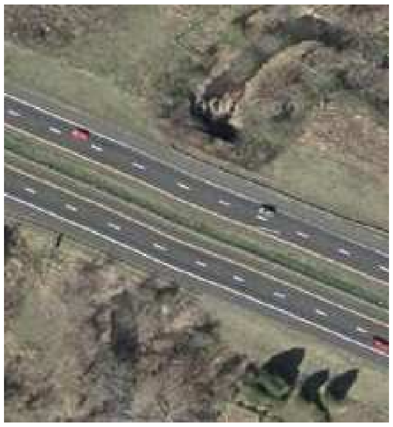


Figure 2: A drainage pipe under a roadway, corresponding to Feature 1 from Figure 1. Picture from Google Maps.

- Feature 6 is a set of small roadways, possibly with bridges.
- Feature 7 is an elevated interchange with a stream flowing underneath it. The exact pattern of flow is difficult to discern from lidar and satellite maps, but it is clear that part of this structure needs to be cut.
- Feature 8 is a roadway over an obvious depression. The sharpness of the cutoff between the road and the surrounding terrain indicates that the road is likely to be raised above the ground prominently here.
- Features 9 and 10 represent areas where a roadway seems to have been completely wiped out by the river, probably in the interpolation step. These are bridge-like features, but our algorithm should ignore them in the end.

Figure 3 shows the results of filtering our data set and grouping the selected locations using the built-in visualization tools in the GRASS software package. The algorithm clearly labels the large uncut bridge-like features in the image, such as the interchange and the drainage pipe, and avoids several pre-cut bridges. It correctly identifies the uncut bridge labeled 4, above, as a noteworthy feature, and isolates several small bridges in the tangle of elements labeled hand-labeled as feature 6. Features 9 and 10, already deeply cut, are ignored. Overall, the computer-generated map seems to capture all of the relevant features except for a few ambiguous parts of area 6, while ignoring already-cut bridges and generating fairly little noise.

Performing the feature extraction on this relatively small (609180 cells) map took 6m 45s. We expect computational complexity to be linear with respect to the number of cells. This bears out in practice: it takes 25 minutes to process a 21117520-cell grid with our system.

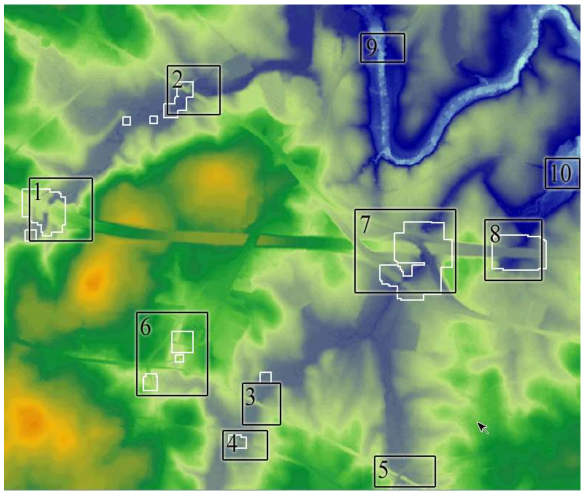


Figure 3: A comparison of hand-labeled (dark) and machine-detected (light) features.

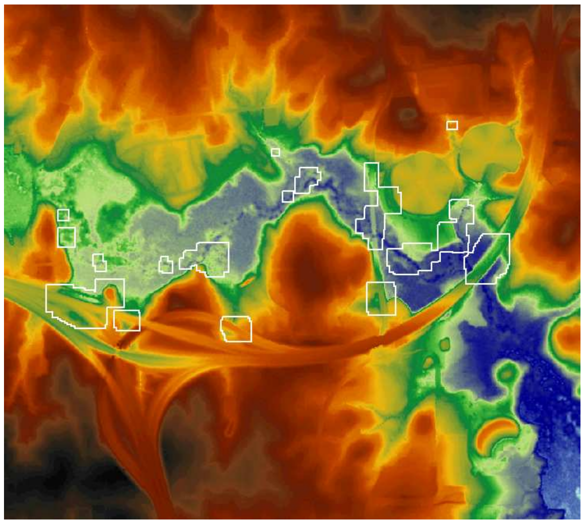


Figure 4: DEM of a road over a ravine in an urban area, demonstrating the shortcomings of the algorithm. Several bridge-like structures are correctly labeled, but these are also many false positives caused by trees in the ravine.

Figure 4 shows a less functional job. This time, the algorithm has identified a series of noisy-looking areas along what looks like a stream as being bridges, as well. Analysis of the image area reveals that the area isn't conventionally filled with water, however: the grainy lumps that have been labeled as bridge-like objects that impede the flow of water are actually trees in a ravine. While the image of all those areas being selected as good areas is rather unsightly, most represent terrain artifacts that can

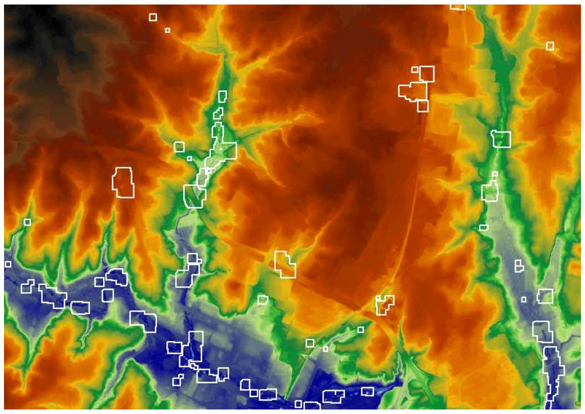


Figure 5: The bridge detection algorithm applied to a larger area. Note the tendency to over-select low areas when they are not uniform.

easily be cut; those that don't are actually major features.

Figures 5 and 6 demonstrate similar results with larger data sets. There are quite a few false positives overall, but numerous bridges are correctly detected.

4 Conclusion and Future Work

Overall, our algorithm seems quite effective in detecting bridges and similar features that impede flow routing on high-resolution DEMs.

Experimenting on other data sets, however, revealed that the algorithm is fairly sensitive to input error. While noise mostly just impairs its ability to detect useful features, errors that produce regular patterns will often lead to numerous false positives. This problem occurs because our program is searching for regular, mostly-linear features, which can be introduced into the image as artifacts during the various stages of preprocessing if the raw data is sufficiently poor. It should be noted that, while such results include a lot of false positives, most of those are clustered around image artifacts, so cutting through such areas shouldn't deform the actual map very much. Very few false positives generated by our algorithm are actually objects that would greatly affect the hydrological model if cut.

The computation time currently leaves something to be desired, however. While the integral image method speeds up the first few statistical filtering steps, we currently use naive techniques to find local extrema – these cost us a lot of time spent rescanning the same pixels as the window moves across the map. Finding the extreme value for a strip of data at a time and then simply taking the extrema of those could greatly speed up the execution of this stage. For the shape filters, a more computationally-efficient way to perform the required rotations would be ideal. Overall, the computational overhead of running the algorithm could probably be significantly reduced by running it as part of another window-sweeping algorithm and reimplementing it in C/C++.

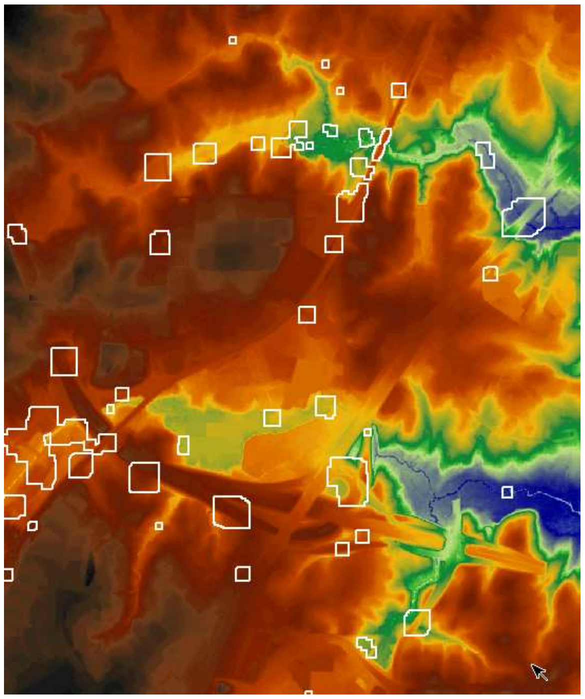


Figure 6: The bridge detection algorithm applied to a large area with a very complex road network. While the system is incapable of puzzling out the interchange, it does identify a large number of bridge-like features (and a few false positives).

It may be possible to get improvements in accuracy by running this algorithm iteratively with a bridge splicer, recalculating the flood fill depth after the removal of the current target bridge. Such a system would require a lot of repetitive computation, however.

Since most of our false positives seem to come from artifacts in the DEM, we believe that simply cutting the regions identified by our algorithm, even if the detected feature is not a bridge, will improve flow-routing.

References

G. Sithole and G. Vosselman. 2006. Bridge detection in airborne laser scanner data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 61(1):33–46, October.

Paul Viola and Michael Jones. 2002. Robust real-time object detection. *International Journal of Computer Vision - to appear*.

Flow Routing on Flat Terrains

Taylor Hamilton
Department of Computer Science
Swarthmore College
thamilt1@swarthmore.edu

Giovanna Thron
Department of Computer Science
Swarthmore College
gthron1@swarthmore.edu

Abstract

Most current flow routing algorithms use digital elevation models (DEMs) to construct flow models. In order to successfully use current techniques for flow routing, they flood local minima and then find a way of routing the flow across the flat surfaces. In this paper, we examine an alternative method for computing flow routing on these flooded surfaces that takes into account the original elevation data. Our approach is based upon Dijkstra's single source shortest path algorithm. We set the distance between two adjacent cells to be the elevation of one of the cells. While our results are not yet ideal, altering our distance formula shows promise for improvement.

1 Introduction

A current problem in geographic information systems is the automatic extraction of river networks from a set of elevation data using the method of flow accumulation. Calculating river networks is useful in determining flood insurance zones.

The basic idea for solving this problem is relatively simple: for each elevation point, route flow to the neighbor with the steepest downhill slope. This method is effective as long as there are no local minima. Local minima will be pits or valleys in which the water will get trapped. Ideally, all water should flow to some outlet point at the edge of the grid.

Current algorithms (Jenson and Domingue, 1988; Garbrecht and Martz, 1997; Soille and Colombo, 2003) tend to deal with this problem by flooding the minima until they are all removed. This approach is justified by the assumption that minima are the accidental result of poor sampling in the original data. However, this is not always the case. Many minima are caused by large-scale terrain features, such as a bridge over a river. When these minima get flooded, useful information about the underlying river is lost, as can be seen in Figure 1.

These flooding algorithms create large flat surfaces, which cause a new problem in flow routing. Without a steepest downslope neighbor it is not immediately obvious in which direction the water should flow across the surface.

Several algorithms have been developed that attempt to solve the problem of flow routing over flat terrain. A side effect common to all these algorithms is that they fail to use information about the original terrain with their flat terrain flow routing algorithms. We have developed an algorithm to route flow across flat surfaces that takes into account the original terrain. This provides river networks that more accurately match reality.

2 Related Work

Current algorithms for solving this problem do not produce ideal results. Jenson and Domingue (1988) focussed mainly on flooding and their method for flow routing on flat surfaces was not very involved. For each point on a flat surface, they assigned the flow to be in the direction directly towards the outlet. This resulted in artificial looking river networks

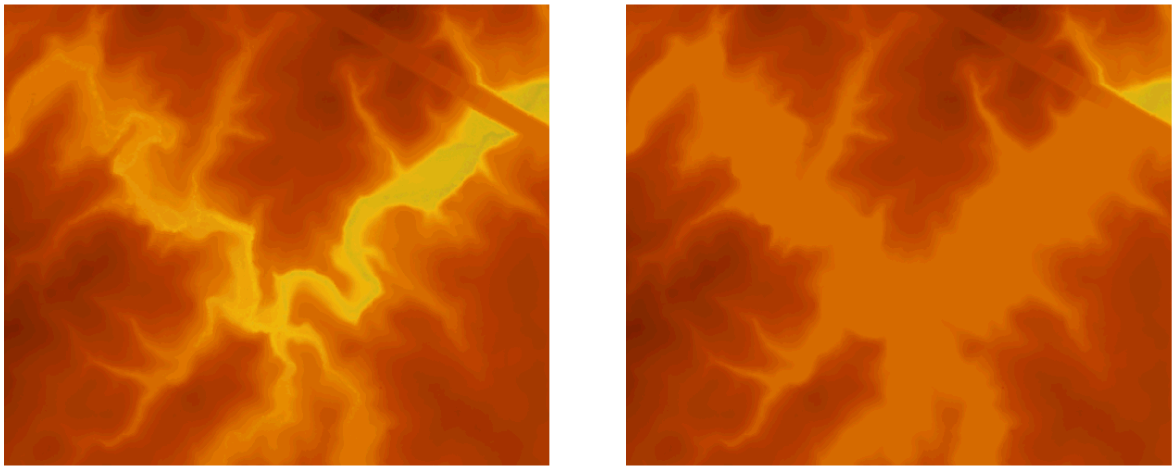


Figure 1: Original and flooded elevation data

because of long stretches of parallel lines.

Garbrecht and Martz (1997) improved upon this algorithm by not only routing flow towards the outlet, but also away from the bordering high terrain. This provided more natural looking river networks. However, as these river networks do not take into account the underlying elevations, they do not always accurately model the true flow of water in the region.

Soille et al. (2003) proposed the method of carving as opposed to flooding for removing minima, which reduces the number of flat areas. They also proposed a flat terrain flow routing algorithm that is an improvement on Garbrecht and Martz's method. Although Soille's algorithm is an improvement, it has the same fundamental issues as that of Garbrecht and Martz.

3 Methods

Our algorithm focuses on improving flow routing over flat terrains. To accomplish this, we use both the flooded terrain information and the original elevation data. The flooded terrain indicates the areas on which to concentrate, and the original terrain provides the elevation data needed for our method. We use Dijkstra's algorithm to calculate the shortest path to the spill points. We vary the metric for computing the distance between two adjacent cells.

We begin with digital elevation models in the GRASS ASCII format: one of the original terrain and one with the local minima flooded. We find the spill points, cells adjacent to the flooded terrain with lower elevations.

We used a single source shortest path algorithm to calculate flow directions for flat areas. This algorithm treats our grid as a connected graph where each cell is connected to its eight neighbors. The weights of the edges can be chosen independently of the algorithm, and we experiment with several options.

To compute the single source shortest path, we used Dijkstra's algorithm. The algorithm begins by initializing all the path lengths of any cell to the spill point to infinity. We create a priority queue that contains the spill points, and set their path lengths to zero. We continue extracting the point from the priority queue with the minimum path length until the queue is empty. Each time we remove a point, we look at each of its neighbors and update their paths if the path through the current point is shorter than the stored path. We then add each updated neighbor to the priority queue.

When the algorithm is finished, the result is a forest that spans the area of interest, where each tree is rooted at a spill point. The leaves of the trees are the points farthest away from the spill points. The path from a node to the root of a tree is the shortest path to a spill point.

We can use these trees to calculate flow accumulation. We imagine that a unit of water falls onto each cell in the grid. Using the flow directions of each cell, we can determine the amount of water that accumulates in each cell. Let p be any cell and $F(p)$

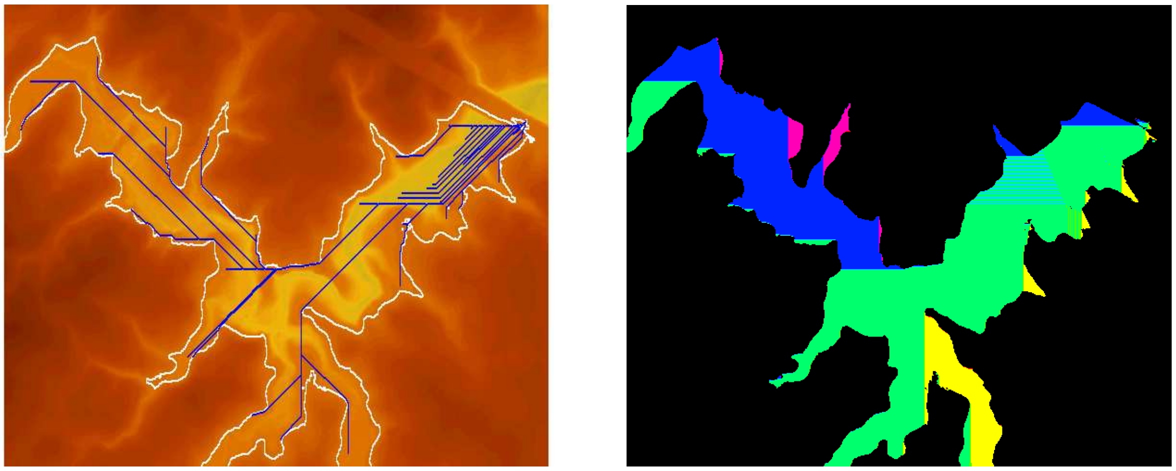


Figure 2: River networks and flow directions using the Euclidean distance metric

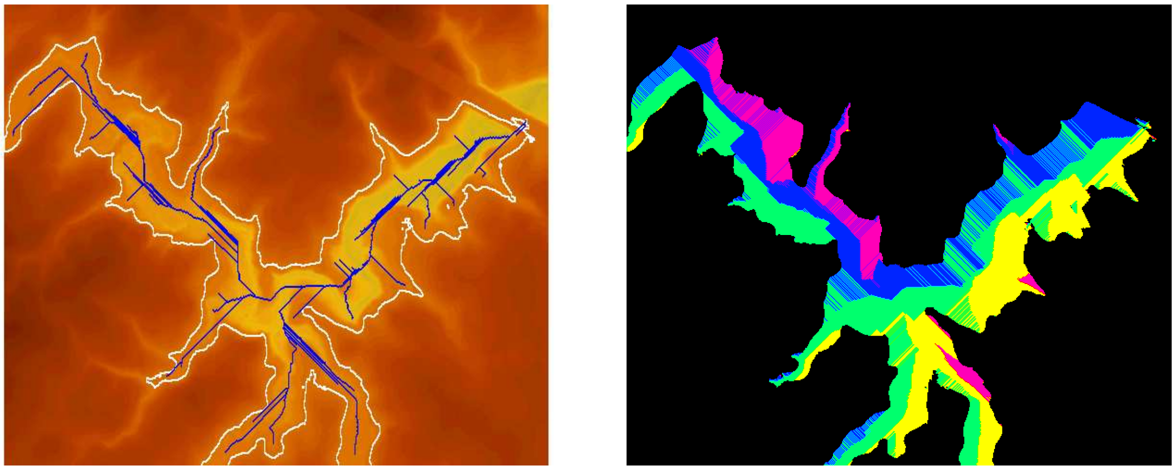


Figure 3: River networks and flow directions using Soille's algorithm

be the set of cells flowing into p.

$$\mathrm{acc}(p) = 1 + \sum_{q \in F(p)} \mathrm{acc}(q)$$

We calculate the flow accumulations of a node in the forest as the sum of the flows of each of its children plus one. A grid showing cells whose flow accumulations are greater than some threshold should show the locations of the rivers of the terrain.

Our algorithm outputs GRASS ASCII files with the flow directions and the flow accumulations at each cell of the grid. We represent flow direction with numbers 1 through 8, corresponding to the eight possible directions of flow. We create river networks based on the set of points with flow accumulations over a given threshold.

3.1 Metrics

Below we present the weights used to determine the distances between cells for the Dijkstra's algorithm. In all cases, we added an extra weight of $\sqrt{2}$ to diagonally adjacent cells to account for the difference in Euclidean distance.

3.1.1 Euclidean Distance

The simplest method we used was setting the distances between adjacent cells to 1. This resulted in the Single Source Shortest Path (SSSP) method, as used by Jenson and Domingue (1988). In effect this method just computes the shortest Euclidean distance from any point to the spill points, and routes flow over that path.

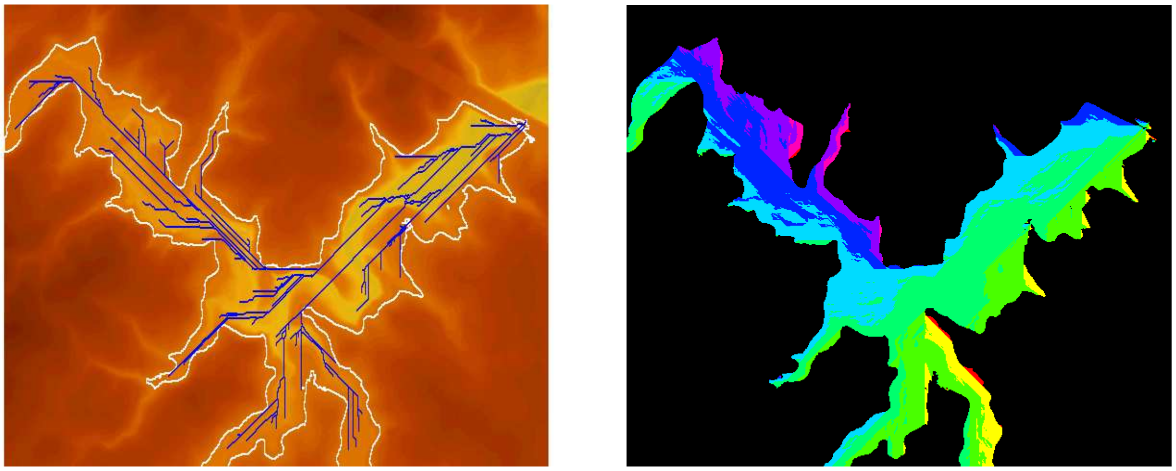


Figure 4: River networks and flow directions using the elevation distance metric

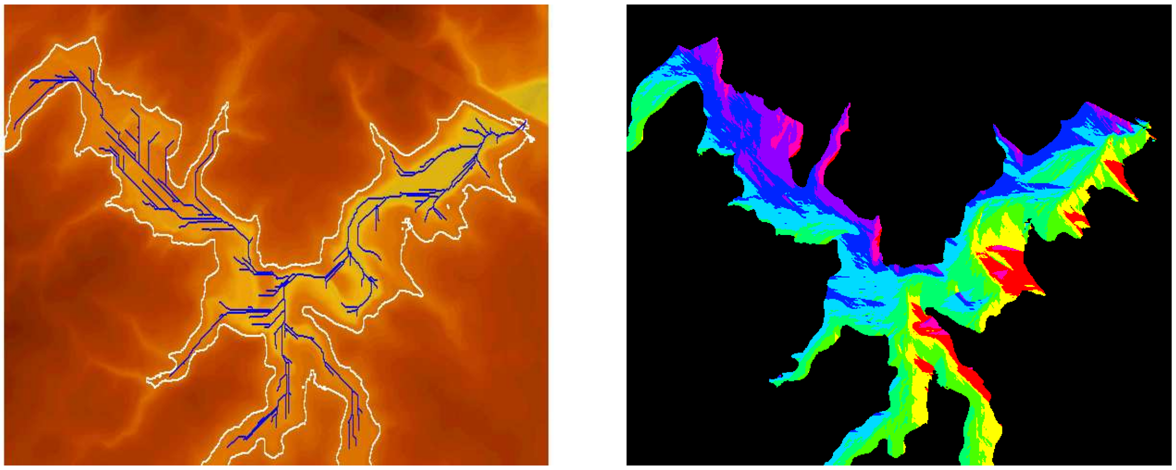


Figure 5: River networks and flow directions using the translated elevation distance metric

3.1.2 Soille's Algorithm

The flat terrain flow routing algorithm introduced by Soille (2003) is designed to route flow through the center of the terrain and avoid having straight parallel lines. We calculate the distance, $d(c)$ from each cell to the border of the flat terrain using a breadth first search away from the border. Let c be a cell and C the set of all cells.

$$w(c) = \max\{d(f) | f \in C\} + 1 - d(c)$$

3.1.3 Elevation

Our first metric that uses the original elevation data sets the distance between any two adjacent cells to the elevation of the cell flowed to. Thus, the total distance from a cell to the spill point is the sum of the elevations of the cells traversed. This encourages the flow to travel down to lower elevations, as well as traversing a small number of cells between the source and the spill points.

3.1.4 Translated Elevation

To weight more heavily the importance of flowing across low elevations, as opposed to traversing short distances, we translate the elevations of all cells down by the minimum elevation over the relevant area. Thus, the weight of a cell is the difference of the elevation of the cell and the minimum elevation.

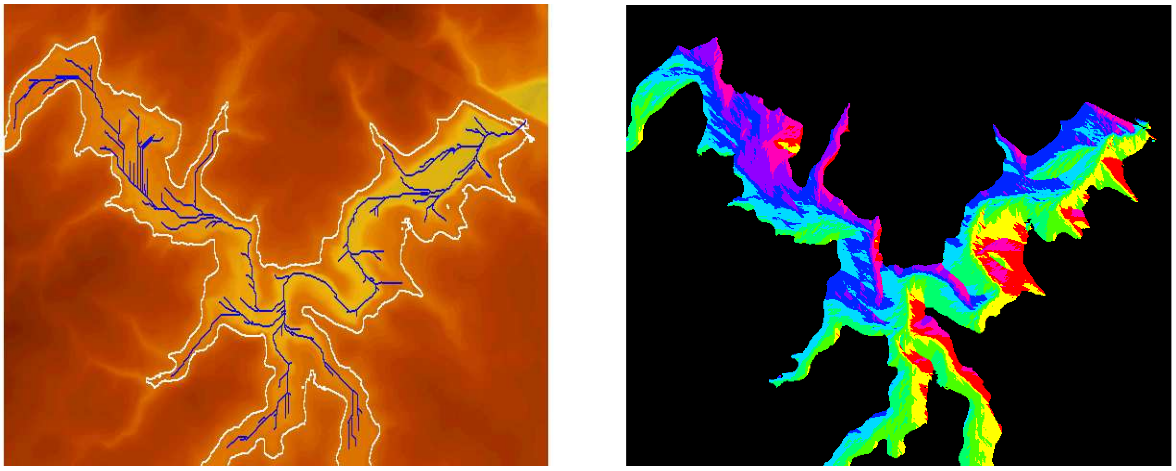


Figure 6: River networks and flow directions using the squared translated elevation distance metric

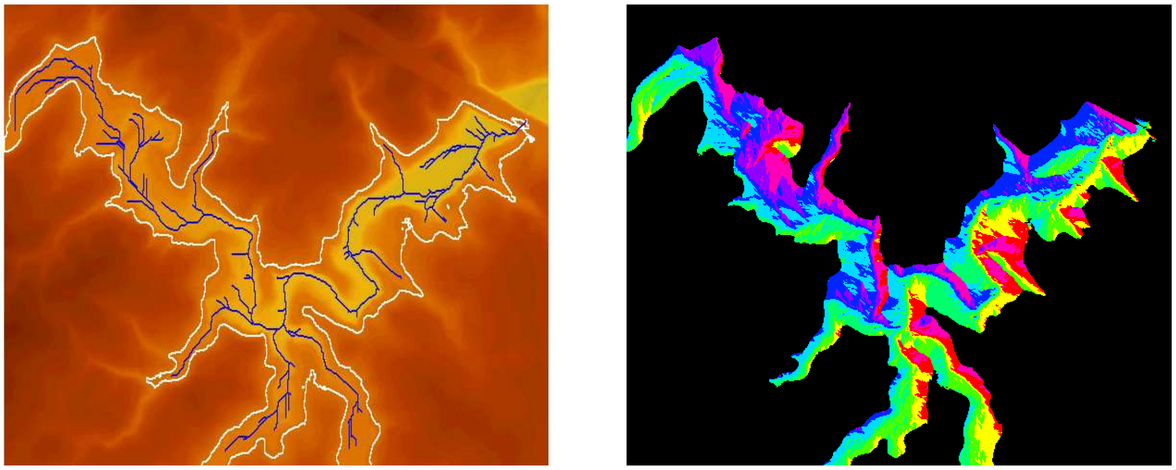


Figure 7: River networks and flow directions using the fourth power of the translated elevation distance metric

3.1.5 Power of Translated Elevation

Raising the translated elevation to a positive power puts a greater penalty on higher elevations. This further encourages flow to follow low elevations. A greater power will put more emphasis on traveling on low elevations.

4 Data

We used raster data of the North Carolina river basin at 10 foot resolution. We used both the original elevation data and elevation data of the terrain with the sinks flooded.

5 Results

For each metric, we show the river networks and flow directions. In computing the river networks, we used an accumulation threshold of 150 cells (150,000 ft^2). In the flow directions figures, each color indicates a different flow direction.

The results of using the Euclidean distance metric are shown in Figure 2, Soille's algorithm in Figure 3, the elevation metric in Figure 4, the translated elevation metric in Figure 5, the squared translated elevation metric in Figure 6, and the fourth power of the translated elevation metric in Figure 7.

6 Discussion

As can be seen in Figures 2 through 7, the results improve with each alteration of our algorithm, eventually producing natural looking river networks that follow the elevations of the original terrain.

By comparison, the Euclidean metric (Figure 2 fails to produce natural looking or accurate rivers. The flow is routed in straight parallel lines and hugs the boundaries of the region, as the algorithm searches for the shortest Euclidean distance.

Soille's algorithm (Figure 3), on the other hand, produces more natural looking river networks. However, as this algorithm fails to take into account the original elevation data, the rivers do not follow the terrain features. As an example of this behavior, observe the oxbow near the center of the region. Rather than following the bend in the river, the river stays in the center of the region.

Each of Figures 5 through 7 shows an improvement on the previous river network. As you can see in Figure 7, our river network both looks natural and accurately models the terrain. Our river network tends to stay in the lighter yellow areas, which corresponds to the lowest elevations in the region.

From the river networks in Figure 7 it can be seen that our algorithm tends to perform better on lower elevations than on higher ones. This is a result of translating the elevations by the minimum elevation of the region. While this translation succeeds in appropriately weighting elevation against Euclidean distance at lower elevations, this balance too heavily in favor of Euclidean distance at higher elevations.

7 Future Work

We would like to develop a distance metric that does not have the drawbacks at higher elevations that our current algorithm displays. To accomplish this, we have experimented with other distance metrics without success. We began by taking the exponential of the elevation, but discovered that this this resulted in numbers that overflowed Python's float type. We would like to experiment with the taking the differences of elevations of neighboring cells.

References

J. Garbrecht and L. Martz. 1997. The assignment of drainage direction over flat surfaces in raster digital elevation models. *Journal of Hydrology*, 193:204–213.

S. Jenson and J. Domingue. 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54:1593–1600.

J. Vogt Soille, P. and R. Colombo. 2003. Carving and adaptive drainage enforcement of grid digital elevation models. *Water Resources Research*, 39:10–1–10–13.

Shapefile Overlay Using a Doubly-Connected Edge List

Phil Katz and Stephen St.Vincent
Swarthmore College
500 College Ave.
Swarthmore, PA 19081
`[pkatz1, sstvinc2]@swarthmore.edu`

Abstract

We present a method for finding the overlays of a set of polygons that uses the doubly-connected edge list structure. We first detect all intersections between polygons using a brute-force method. We then build the doubly-connected edge list, maintaining information about the original polygons, from which we can easily perform shapefile overlay operations: intersection, difference, and union.

Our algorithm runs in $O(n^2)$ time. Our doubly-connected edge list construction algorithm runs in $O(n \log(n))$ time, with the bottleneck being the brute-force $O(n^2)$ line segment intersection. Once that list is built, any given overlay operation is $O(n)$.

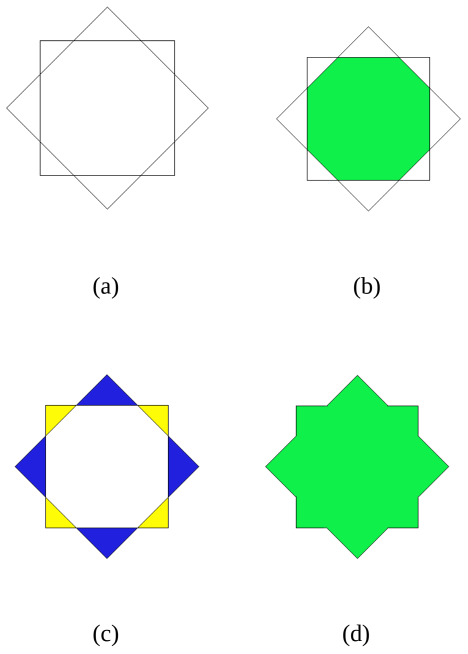


Figure 1: Examples of shapefile overlays. (a) The original polygons in set *S*. Here, we have two overlapping squares at different orientations. (b) The intersection of the two squares, represented by the green region. (c) The difference of the two polygons, shown by the blue and yellow regions. (d) The union of the two polygons. Note that the interior segments are now gone.

1 Introduction

Natural disasters such as floods often occur swiftly and without warning. It is imperative, therefore, that political entities such as counties and states be adequately prepared to deal with such disasters. Often, this level of preparation varies directly with the amount of funding recieved, which is in and of itself a function of the percieved threat in that region. Determining the extent to which region is in danger of flooding is difficult to assess anecdotally. However, by combining geographical data such as watershed layouts with political boundary data, we can use shapefile[1] overlays to assign an unbiased value to the level of danger to any region. This information

[1]A shapefile is a common file format for exchanging polygon data that does not maintain topology

can then assist in appropriate resource allocation, as well as computation of flood insurance rates.

Given a set of polygons *S*, how can one efficiently determine the new set of polygons *P* that is defined by the overlay of the polygons in *S*? Polygons in *P* can include the intersection, union, or difference of members of *S*. Figure 1 shows examples of these types of overlays.

To solve the problem of shapefile overlay, we will use methods from (de Berg et al., 1998) to build a data structure called a *doubly-connected edge list* that will allow us to calculate overlays efficiently. The more general problem of shapefile overlay has special cases which would be unusual in settings such as the one described above, such as polygons with holes in them. Still, we consider these special cases to make our algorithm as general as possible.

In section 3, we present our methods for building the doubly-connected edge list and calculating shapefile overlays. In section 4, we discuss the runtime analysis of our algorithms. In section 5, we present results applying our algorithms to simple test-case polygons as well as to real geographical data. Finally, in section 6, we discuss the implications of our results.

2 Related Work

When building topological representations, it is usually the case that we wish to restrict the topologies to follow a specific set of constraints. (Hoel et al., 1994) describe such a system. Their topologies have certain consistency requirements, and as such must follow an explicit set of rules, such as the following:

- Interiors of polygons in a feature class must not overlap
- Polygons must not have voids within themselves
- Polygons of one feature class must share all of their area with polygons in another feature class

Shapefile overlay is necessary to enforce these rules on large sets of polygons. Without an effective shapefile overlay algorithm, the work of (Hoel et al., 1994) could not be implemented in an efficient, robust manner.

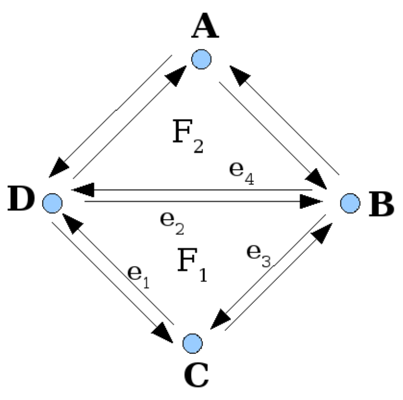


Figure 2: Doubly-connected edge list for two polygons. Half-edge e_1 has a `next` pointer to e_2 and a `previous` pointer to e_3. The current face of all three of these is F_1. The twin of e_2 is e_4, whose current face is F_2. The twins of e_1 and e_3 have `Null` as their current face.

3 Methods

To find the overlays of multiple polygons, we must construct the doubly-connected edge list. Before we can do that, we need to know where the segments of each polygon intersect the segments of the other polygons in the set.

3.1 Line intersection calculations

This is by far the most straightforward step in calculating shapefile overlays. Because the runtime of brute-force algorithms is not significantly slower than more complex algorithms for the data sets we are considering (Andrews et al., 1994), it does not cost us significantly to implement a brute-force algorithm. For each segment of a polygon, we check explicitly whether it intersects any segment of any other polygon, and keep a list of all intersection points that occur on that segment. This allows us to easily create all of the subsegments for the doubly-connected edge list, as well as to keep track of which of the original polygons a segment was associated with.

3.2 The doubly-connected edge list

A doubly-connected edge list (de Berg et al., 1998) stores all of the information regarding the set of polygons that is necessary to calculate the shapefile overlays. The basic setup of a doubly-connected edge list begins with the edges. The edges of each polygon are stored as directed *half-edges* that go around the polygon in clockwise order such that the face that is bound by the half-edges is always to the right of each half-edge (see figure 2).

Each half-edge stores the following information:

- Starting point
- Ending point
- The ID of the face from which the half-edge originated
- A pointer to the next half-edge on the current face
- A pointer to the previous half-edge on the current face
- A pointer to its twin half-edge

As we build the half-edges that are on the interiors of the original polygons, we can build their *twin edges*. Twin edges are the same as the original half-edge, but with its orientation reversed and its face set to `null`. So, for half-edge *e*, `twin(twin(`*e*`))`=*e*.

For each new half-edge that we add, we update a dictionary, `vDict`, that contains all of the necessary edges. The `vDict` is a hash table keyed on vertices and contains a list of the half-edges that start at that vertex. This is critical for building the next- and previous-edge pointers.

3.2.1 Creating next- and previous-pointers

Determining the next and previous pointers for each half-edge is non-trivial. For the next pointer of each half-edge *e*, we must find the half-edges whose starting point is the same as the ending point of *e*. Then, we must determine which of these half-edges makes the largest clockwise angle with *e*, and set the next-pointer of *e* to that half-edge.

Now that the `vDict` has been updated for each half-edge, this task becomes much easier. Using figure 3 as an example, let us attempt to find the next-pointer of the half-edge $\overline{AX}$. By treating each half-edge as a vector originating at X , we can find the angle between the half-edge $\overline{AX}$ and all the other half-edges by using the cross product and the dot product:

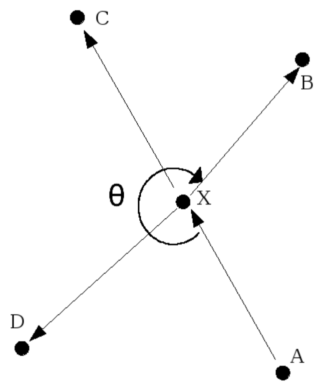


Figure 3: Finding the next-pointer. Here, we are trying to find the appropriate half-edge for the next-pointer of half-edge $\overline{AX}$. θ measures the angles between $\overline{AX}$ and the other half-edges in the image. We choose the half-edge with the largest value of θ, which in this case is $\overline{XB}$.

$$\sin(\theta) = \frac{\left|\left|\overline{AX} \times \overline{BX}\right|\right|}{\left|\left|\overline{AX}\right|\right| \left|\left|\overline{BX}\right|\right|}$$

$$\cos(\theta) = \frac{\overline{AX} \cdot \overline{BX}}{\left|\left|\overline{AX}\right|\right| \left|\left|\overline{BX}\right|\right|}$$

After computing $\cos(\theta)$ and $\sin(\theta)$, we can calculate the true angle θ (where $0 \leq \theta \leq 2\pi$) between $\overline{AX}$ and each of the other three half-edges in the figure. The candidate half-edge with the largest value of θ can now be set as the next-pointer. To complete the example, the next-pointer of $\overline{AX}$ would be $\overline{XB}$, and the previous-pointer of $\overline{XB}$ can be set to $\overline{AX}$. After walking completely around a face in this fashion, all of the next- *and* previous-pointers for the half-edges that bound that face will have been set, so we do not need to calculate the previous pointers explicitly.

3.3 Computing the overlays

Now that we have walked along the half-edges of every face, we have a list for each new face of all of the original faces from which it was derived. From here, we can easily specify an overlay by selecting the new faces that meet the criteria of the overlay in question.

3.4 Non-intersecting overlays

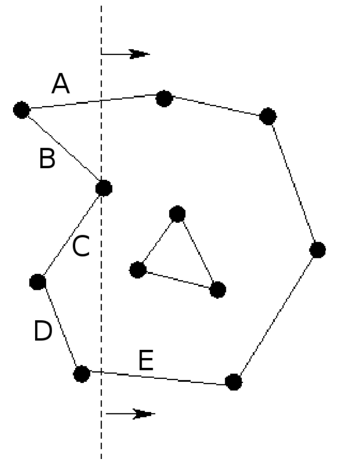


Figure 4: Line-Sweep Algorithm. The dotted line is sweeping rightward. Right before reaching the current point, there should be four edges (A,B,C,E) in the data structure, and upon reaching this point, two (B,C) should be removed.

Not every overlay of two polygons will involve the inteserction of their segments. Consider for instance the overlay of the Sahara Desert with Cumberland County, Pennsylvania. The intersection of these two polygons should be null, given that the two polygons are clearly not spatially coincident and have no intersections.

But we cannot simply say that a lack of segment intersections implies a lack of spatial coincidence. Consider now the overlay of Nebraska with the entire United States. While it is not apparent why one would choose to perform this overlay, it should be clear that this is an example of an overlay that has no segment intersections but does have some overlap.

We solve this problem by using a line-sweep algorithm (de Berg et al., 1998). Figure 4 shows two polygons, with one completely interior to the other. Once our line-intersection algorithm determines that there are no segment intersections between these two polygons, we can move into our line-sweep algorithm. We sort the vertices in order of their x-coordinate; we also store the face associated with the first vertex in the sorted list. We then step through the sorted vertices in order while maintaining a list of edges that currently intersect the sweep line. When a vertex is encountered that has an associated face that differs from the face of the first point, we can stop our line-sweep. If the new vertex is below an odd number of segments, then we know that the new face is completely interior to the other; otherwise, it must be completely exterior. This method is robust for concave polygons.

3.5 Polygons with holes

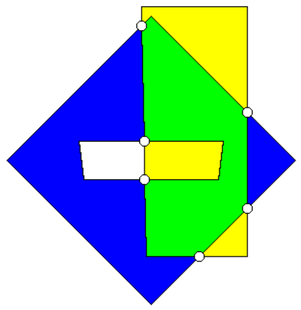


Figure 5: Polygons with holes. The blue polygon has a hole in its center, which is not filled in. Where the yellow polygon intersects with the hole, it remains unchanged.

Consider the nations of South Africa and Lesotho. Lesotho is an independent nation completely interior to the borders of South Africa. As such, South Africa may be treated as a polygon with a hole in it. If we worked for the South African government and needed to determine overlays, we would surely want to take Lesotho into account.

We can handle this by allowing each face to have a pointer to its *inner edges*. These inner edges will form a closed polygon. The outer half-edges of this interior polygon (which run counter-clockwise) have the same face as the original polygon, while the inner half-edges (which run clockwise) have their face set to null. From here, we can perform our normal shapefile overlay process, starting with segment-intersection.

4 Runtime analysis

Our brute-force polygon intersection calculation is $O(n^2)$. For each line segment, we test all of the line segments in the other polygon for intersection. Given that we do not check segments in the same polygon, this upper bound of $O(n^2)$ can never be reached, but is still the appropriate theoretical upper bound. This assumes that the polygons are simple.

Our line-sweep algorithm is $O(n \log(n))$. For each of the n vertices, the operations we need to perform (search, insertion, and deletion) can be implemented (with a binary tree, for example) to require $O(\log(n))$ operations to maintain the line-sweep data structure.

Once we have the doubly-connected edge list built, we will have k edges, stemming from the n original edges. To build the overlay faces, we need only go through each of the k edges once, removing them from the list of all edges once we assign them to a face. So the overlay algorithm itself (assuming that the doubly-connected edge list has already been built) is $O(k)$. In the worst case, k is $O(n^2)$, but in most real-world polygon intersections, k will be $O(n)$.

5 Results

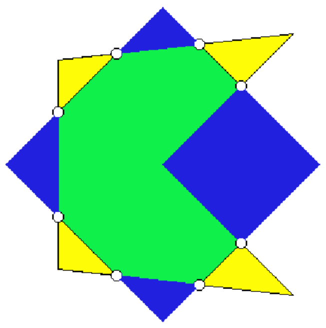


Figure 6: Example of polygon intersection. The blue area indicates areas that are only covered by the convex polygon (rotated square). The yellow areas were only covered by the concave polygon. The green area represents the intersection of the two polygons. The white circles show points of intersection between the segments of the two polygons.

Figure 6 is a sample run of our intersection algorithm on two hand-made polygons. The image was created using the Python graphics library from (Zelle, 2004). The green polygon in the center shows the intersection polygon. The blue and yellow polygons combine to define the difference of the polygons. The entire shaded area is the union of the two polygons. The small white circles show the points of intersection between the two polygons.

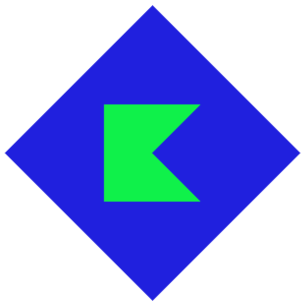


Figure 7: Interior polygons. This image shows the effects of scaling the concave polygon in figure 6 so that it fits completely inside the concave polygon. No area has maintained the yellow coloring from above.

Figure 7 shows the intersection of two polygons where one of the polygons is completely interior to the other. Despite the lack of intersection points, our algorithm has handled this flawlessly.

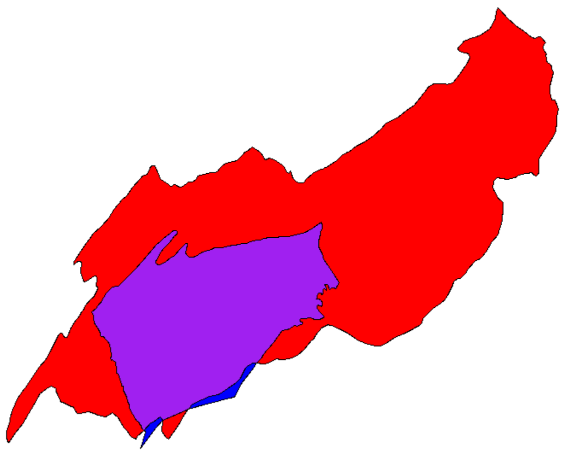


Figure 8: Intersection of Cumberland County with watershed 2050305. The red region is the watershed only, the blue region is Cumberland County only, and the purple region is the intersection of the two.

Figure 8 shows the intersection of Cumberland County, Pennsylvania, with watershed 2050305 (USGS Hydrological Unit Code). Figure 9 shows Cumberland County intersected with watershed 2050306.

Polygons	Time (ms)	Points	Intersect time (ms)	DE list time (ms)	Overlay time (ms)
CC, WS5	8126	693	6132	238	1642
CC, WS6	7950	688	6000	252	1556

Table 1: Benchmarking on sample county and watershed data. The Cumberland County (CC) polygon had 284 points; the watershed 2050305 (WS5) polygon had 409 points; the watershed 2050306 (WS6) polygon had 404 points.

.

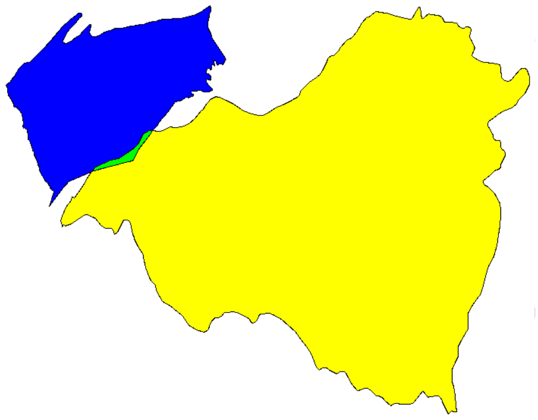


Figure 9: Intersection of Cumberland County with watershed 2050306. The yellow region is the watershed only, the blue regions are Cumberland County only, and the green region is the intersection of the two.

Table 1 shows benchmarking figures for these two runs. The tests were run on an Intel(R) Pentium(R) 4 running at 3.00 GHz with 156 MB of RAM.

6 Discussion

The goal of this project was to implement shapefile overlay in an efficient manner. In order to accomplish this goal, we implemented a doubly-connected edge list that allowed our algorithms to efficiently compute overlays. With the exception of our brute-force line segment intersection, all of our algorithms are relatively computationally inexpensive. Table 1 shows that the line segment intersection is the clear bottleneck. It is possible to implement faster line segment intersection algorithms, but that is not within the scope of this project.

Our algorithm works successfully for most complex cases, including concave polygons, polygons inside other polygons, polygons with holes, and up to three polygons, as in figure 10. Our algorithm can not handle the case of intersecting two polygons that share an edge. This case is pathological for line intersection as well as for constructing a doubly-connected edge list.

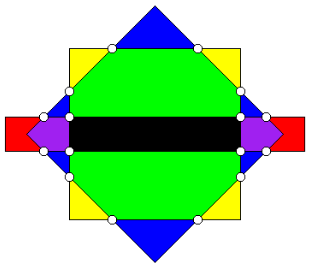


Figure 10: Intersection of Three Polygons. This image shows that our algorithm can be extended to three polygons.

7 Conclusions

Shapefile overlay is a fundamental building block of computational geometry. It is imperative to the field that shapefile overlay can be computed quickly and correctly. Although we do not introduce any revolutionary methods to accomplish this task, we show that it can be performed with fairly straightforward algorithms on simple data structures.

References

D. S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. 1994. Further comparison of algorithms for geometric intersection problems. In *6th International Symposium on Spatial Data Handling*.

Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1998. *Computation Geometry: Algorithms and Applications*. Springer.

E. Hoel, S. Menon, and S. Morehouse. 1994. Building a robust relational implementation of topology. In *8th International Symposium on Spatial and Temporal Databases*.

John Zelle. 2004. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle, & Associates.