Finding shortest paths on large meshes

Vishal Verma

December 17, 2008

Abstract

Finding shortest paths and distances on a mesh is a well studied problem. Algorithms for solving this problem have mainly concentrated on optimizing the time requirement. However for large meshes of the kind we encounter in GIS, the major bottleneck is the memory requirement. This report evaluates different algorithms for computing shortest paths on triangulated manifolds. We adapt these algorithms to work for large meshes, *i.e* meshes that cannot fit into memory. For the "single source, all destinations" problem we use the Dijkstra's [1], fast marching [2] and MMP [3] algorithms. For "single source, single destination" we use A-star heuristics [4] with the Dijkstra and MMP algorithms.

1 Introduction

In this report we evaluate several methods for computing shortest paths and adapt them for large meshes. Given a source vertex on the mesh, these methods compute a distance function on the vertices of the mesh (as in Dijkstra's method and fast marching method) or on all points on the edges of the mesh (as in the MMP algorithm). This distance function can be later used to trace back the shortest path (or an approximation to it) to the source. The major issue in implementing these algorithms is that for large meshes the memory requirement is very high. Given a mesh with n vertices, the memory requirement for computing distances on this mesh using Dijkstra's method or fast marching method [2] is $O(n)$, while for MMP algorithm [3] it is $\Theta(n^{3/2})$. For large meshes, this amount of memory is not available and straightforward implementation of these methods will result in many cache misses at run time.

Unlike Delaunay triangulation or smoothing, the shortest path problem is not a local problem. In a smoothing algorithm or Delaunay triangulation it is sufficient to keep only a small neighborhood of a vertex/triangle in memory to process that vertex/triangle. This is not the case for shortest path algorithms. For example, the shortest paths through a triangle may change if a ridge is added to the mesh someplace very far from this triangle. Due to the non-local nature of this problem, reducing the memory requirements for shortest algorithms is more difficult than for problems that are inherently local.

The rest of this report is divided into three sections. In section 2 we discuss the streaming framework we use to process large meshes. Section 3 looks at three different methods for solving "single source, all destination" shortest path problems. In this section we also describe how to adapt these algorithms to the streaming framework. In section 4 we look at A-star shortest path search algorithm of solving the "single source, single destination" shortest path problem. We also propose a scheme which allows us to use A-star heuristics with the MMP algorithm. In section 5 we look at two reordering schemes, which help reduce the memory requirements of the various shortest path algorithms.

2 Streaming Mesh Representation

A mesh in the streaming format is a sequence of points, triangles and "finalize vertex" tags. The sequence maintains two invariants:

1. A point appears in the sequence before any triangle that uses is it a vertex
2. After a "finalize vertex" tag has been read from the stream, no triangles read from the stream will contain that vertex.

In this format we can begin processing a part of the mesh before reading in the whole mesh. The streaming format works very well for local computations, where we start processing points and faces as soon as

they are read from the stream and can finish processing them, output the results and remove them from memory as soon as we read the finalize tag for that vertex.

Shortest path algorithms are inherently non-local operations, but Dijkstra-style shortest path algorithms have certain properties which make them suited for the streaming framework. In this report we will be considering only Dijkstra-style shortest path algorithms.

3 Single source, all destinations shortest path algorithms

Dijkstra's algorithm forms the base of many "single source, all destinations" shortest paths algorithms. The three algorithms we have implemented for this problem: Dijkstra, fast marching and MMP, use a Dijkstra-style mesh traversal. Below we present the general structure of a Dijkstra-style mesh traversal algorithm. The primary data structure used in this method is a priority queue, Q, containing vertices that are sorted on their distance values.

procedure *Generic_Dijkstra*

```
1: Initialize distance value of all vertices to ∞
2: Set the distance value of source to 0
3: Mark all vertices except source as "unseen"
4: Insert the source vertex into Q
5: while Q is not empty do
6:    v = Q.pop()
7:    mark v as "done"
8:    for all neighbors u of v do
9:       if u is not marked "done" then
10:         update the distance value of u for the path
            through v
11:         if u is "unseen" then
12:            insert u in Q
13:         end if
14:      end if
15:   end for
16: end while
```

At a given time during the algorithm we have three types of vertices(regions). See figure 1.

1. The inner white region is the set if vertices which have been marked "done". The distances to these vertices from the source have been computed and will not change through the remainder of the algorithm.

2. The set of vertices in Q forms the grey "frontier". Some distance values to these vertices from the source have been computed. But these values may decrease in the course of the algorithm.

3. The black outer region is the set of vertices marked "unseen". These vertices have not been visited by the algorithm.

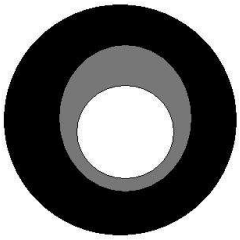


Figure 1: Three different regions during a Dijkstra like mesh traversal

To adapt such an algorithm to the streaming format we make use of the following observations:

1. The "frontier" should be in memory.

2. Those "unseen" vertices which are not neighbors of a frontier vertex need not be kept in memory

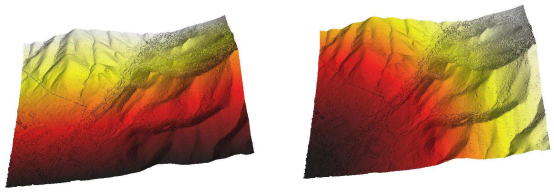
3. Those "done" vertices that are not needed for the update step of the algorithm can be ejected from memory.

4. We would like the "done" vertices to appear in stream before the "frontier" vertices, which in turn should occur before the "unseen" vertices.

Thus, ideally at any point in the algorithm, we would like to keep only the frontier and its immediate neighborhood in memory. To achieve this, we reorder the mesh in a spiral sequence (in terms of the x, y coordinates of the vertices) around the source. See figure 2(b). The vertices are colored in the order they are read from the stream. Vertices appearing earlier in the stream have a darker color, while those appearing later have lighter color.

In the average case, the size of the frontier region is $O(\sqrt{n})$, where n is the number of vertices in the mesh. Thus, with a good reordering scheme we can expect the memory requirement to fall to $\Theta(\sqrt{n})$.

In the following sections we discuss implementation of three different Dijkstra-based algorithms with an



(a) Mesh with source at the lower left corner (b) Mesh reordered around source

Figure 2: Reordering of a streamed mesh

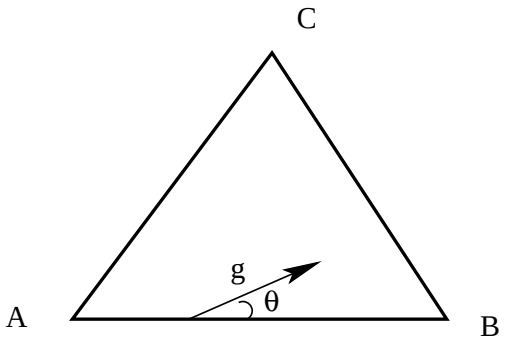


Figure 3: Computing approximate shortest distance to C, using shortest distance values for A and B

aim to reduce their memory requirement using the above ideas.

3.1 Dijkstra's Method

The standard Dijkstra's algorithm finds shortest paths over the edges of the mesh. In this algorithm the update step 7 of *Generic_Dijkstra* does not use any of the "done" vertices. Thus we can discard a vertex from memory as soon as it is marked "done". We modify the standard Dijkstra's method to take advantage of the above observation.

procedure *Dijkstra*(*streamed_mesh*, *source*)

```
1:  reorder streamed_mesh
2:  read from streamed_mesh until source vertex is
    read
3:  source.dist=0
4:  insert source into Q
5:  while Q is not empty do
6:    v=Q.top()
7:    if v has been finalized then
8:      Q.pop()
9:      mark v as "done"
10:     for all neighbors u of v do
11:       if u is not marked "done" then
12:         update the distance value of u for the
            path through v
13:         if u not in Q then
14:           insert u in Q
15:         end if
16:       end if
17:     end for
18:     output necessary information for v and then
        discard v from memory
19:   else
20:     read from streamed_mesh
21:   end if
22: end while
```

Results

We ran the above algorithm on a mesh with five million triangles. On the reordered mesh (figure 2(b)) the algorithm took 35 Mb of memory and 93 seconds to run, while on the unordered mesh (figure 2(a)) it took 321 Mb of memory and 95 seconds to run. We can see that reordering does help in saving memory.

3.2 Fast marching

Fast marching is an algorithm for tracking the movement of a growing wavefront on a mesh (based on *Eikonal equations*). The method computes the approximate time when a wave that starts at the source hits a vertex of the mesh. Computing geodesic distances on a mesh is a particular case of this problem in which the wavefront advances at a unit speed. This algorithm can also be used to compute weighted shortest paths. Below is a brief explanation of the algorithm; for details refer to [2].

Fast marching differs from Dijkstra's algorithm in its update step. Let B be the vertex being popped from the queue containing the "frontier" and let C be any neighbor of B which has not been marked "done". Let A be a common neighbor of B and C. These three vertices form a triangle ABC; see figure 3.

Let the current distances of A, B and C be T_A, T_B and T_C. If vertex A is not marked "done", then we compute the candidate distance at C by

$$d_C = T_B + |BC|$$

otherwise, we assume that the gradient, g, of the distance function is constant over the triangle ABC. We

compute

$$\theta = \arccos\left(\frac{T_B - T_A}{|AB|}\right)$$

.

We use the direction of the gradient to compute to compute candidate distance to C.

$$d_C = T_A + |AC|\cos(\angle A - \theta)$$

If C is unseen or d_C is less than T_C, we set $T_C = d_C$. If C is unseen we push it into the frontier queue.

Note that for the update step we need to keep in memory those "done" vertices which are neighbors of vertices in the frontier. Thus a vertex can be discarded from memory only if all its neighbors are marked "done". The streaming version of the fast marching algorithm that accommodates the above observation is as follows.

procedure $Fast\_Marching(streamed\_mesh, source)$

```
1: reorder streamed_mesh
2: read from streamed_mesh until source vertex is
   read
3: source.dist=0
4: insert source into Q
5: while Q is not empty do
6:    v=Q.top()
7:    if v has been finalized then
8:       Q.pop()
9:       mark v as "done"
10:      for all neighbors u of v do
11:         if u is not marked "done" then
12:            update the distance value of u for the
               path through common triangles of u and
               v
13:            if u is "unseen" then
14:               insert u in Q
15:            end if
16:         end if
17:      end for
18:      for all neighbors u of v do
19:         if all neighbors of u are marked "done"
            then
20:            output necessary information for u and
               discard u from memory
21:         end if
22:      end for
23:   else
24:      read from streamed_mesh
25:   end if
26: end while
```

Results

We ran the above algorithm on the reordered mesh (figure 2(b)). It took 15 Mb of memory and 100 seconds to run. The same algorithm when run on the unordered mesh (figure 2(a)) took 173 Mb of memory and 101 seconds to run. Again we see that reordering helps us reduce the memory requirement.

3.3 MMP algorithm

The MMP algorithm computes exact geodesic distances on a triangulated manifold. Unlike Dijkstra's method, these distances correspond to shortest paths that can pass through the interior of triangles. Consider a shortest path on the manifold, which passes through a sequence of faces. If we unfold this sequence of faces onto a plane, this shortest path will correspond to a straight line. In general, the shortest paths do not pass through vertices. The exceptions to this are saddle points and corner vertices. These shortest paths can be visualized as rays originating at the source and satisfying the following properties:

1. When passing through the interior of a triangle, the ray is a straight line.

2. When crossing an edge from one triangle to another, the ray bends in such a way that on unfolding the two adjacent triangles onto a plane, the ray corresponds to a straight line.

3. Mitchell [3] prove that geodesics can pass through vertices only where the total angle is greater or equal to 2π, *i.e.* at saddle points. If the mesh has corners (some non-manifold meshes), shortest paths can pass through corner vertices as well. At vertices where the total angle is greater than 2π, many straight directions are possible for an incoming ray. See figure 4. Thus, these vertices act like sources from which a set of rays originate, although only in a small range of directions. Corner vertices also show a similar property. Such a vertex is called a pseudosource. We discuss them later in this section.

The key idea of MMP is to track a set of shortest paths together in a data structure called *window*. A window stores the distance function on an interval of an edge of the mesh. The MMP algorithm partitions edges into intervals such that the shortest path to each point in the interval passes through the same sequence of triangles and pseudosources. Let w be a window and s' be the closest pseudosource to w in

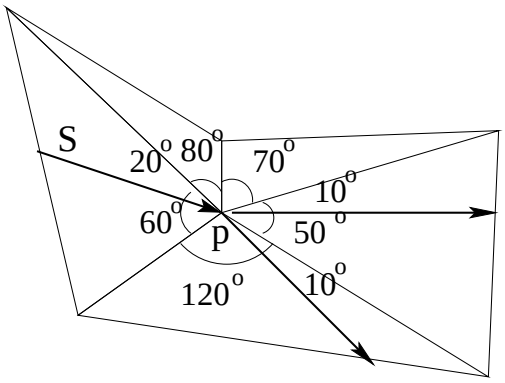


Figure 4: Shortest paths through a saddle point

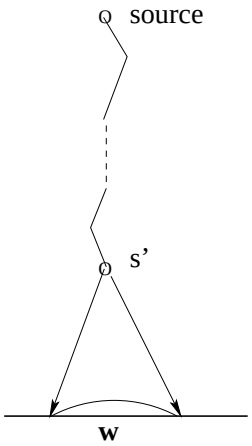
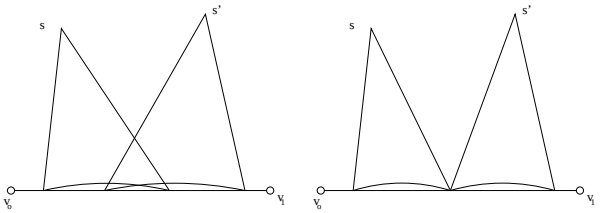


Figure 5: Shortest paths to w through pseudosource s'



(a) Two overlapping windows (b) Partitioning them into two disjoint windows

Figure 6: Handling window overlap

the sequence of triangles and pseudosources, which the window w corresponds to. We assume that we know the distance to s' from the source. Thus, to compute the distance from source to w, we need to only compute the distance from s' to w. Consider the sequence of triangles after s' in the sequence of triangles and pseudosources corresponding to w. On unfolding this sequence of triangles, the shortest paths from s' to points in w appear as straight lines from the s' to points in w. See figure 5. To keep track of these shortest paths, we just need to keep track of the position of the pseudosource in this unfolding. This is done by storing the distance values at the endpoints of w in the corresponding window.

To retrieve the distance values for points in w, window w contains the following information:

- The end points of the interval it corresponds to.
- The distance value of these end points from s'
- The distance of s' from the source. If there is no pseudosource in the path, this value is 0. We call this distance the *source distance* of the window.
- A boolean that signifies which side of the edge does s' lie on.

3.3.1 Window propagation

Window propagation is a step by which the distance function of a window is used to create potential windows on the edges of the adjacent face. See figure 7. To propagate a window, we use the distance values at the end point and the boolean flag stored in this window to compute the location of the pseudosource in this unfolding. The intervals of these potential windows may overlap with those of other windows on that edge. The points in these intervals of overlap have two sets of paths from the source; we need to choose the shortest path among these two sets for each point in the interval of overlap. We do so by subdividing the interval of overlap into sub-intervals such that shortest paths to all points in a subinterval belong to the same of set of shortest paths. Figure 6(a) shows two overlapping windows whose intervals are repartitioned such that the new windows (as shown in figure 6(b)) are disjoint and each point in the new windows has a distance value which is the minimum of the distance value at that point as computed from the two original windows in figure 6(a).

3.3.2 Propagating vertices

In addition to windows, the distance value at some vertices also must be propagated to create potential windows on the adjacent edges. We call these *critical vertices*. There are three types of such vertices:

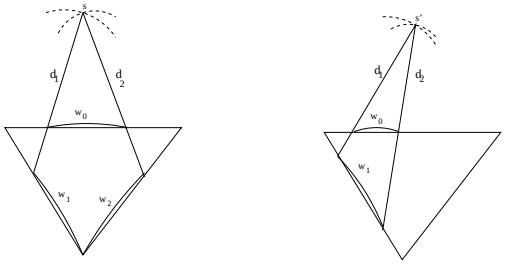


Figure 7: Two different cases in window propagation

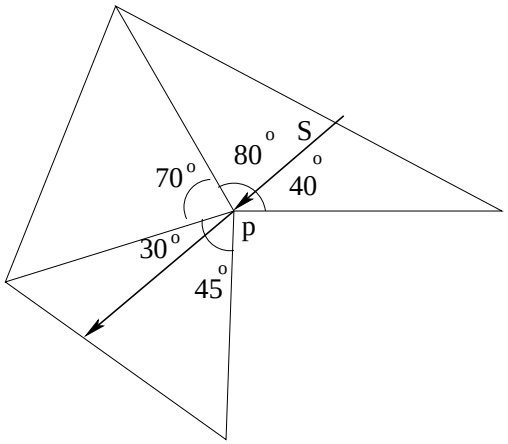


Figure 8: Shortest paths through a corner vertex

- **Source:** The distance value at source is propagated in all directions, as shortest paths from the source to points on adjacent edges are just straight lines. To propagate the distance value at source we create windows on the edges adjacent to the source. These windows span the whole interval of the edge. The source distance of such windows is 0.

- **Saddle points:** Saddle points have a total angle greater than 360^o. In figure 4, S is the shortest path to the saddle point p. The region between the outgoing rays at 180^o and -180^o angle from S cannot be reached by window propagation. So, we propagate the distance value at the saddle point vertex to the adjacent edges in this region. The source distance for such windows is the distance of p from the source.

- **Corner vertices:** In figure 8, S is the shortest path to the corner vertex p. We can see that the region beyond the outgoing ray at 180^o from S will not be reached by window propagation. So, we propagate the distance value at the corner vertex to the adjacent edges in this region. The source distance of such windows is the distance of p from the source.

In figure 5 w can be imagined to be a potential window created by propagating the critical vertex s'. The distance of the endpoints of this window from s' is just their Euclidean distance from s'.

The overlap of these potential windows with existing windows is handled as in the previous section.

3.3.3 The algorithm

The overall algorithm propagates the windows and critical vertices across the mesh in a Dijkstra-like fashion. It uses a priority queue to store windows and critical vertices that lie in the "frontier" region and haven't been propagated yet. We call this queue the *frontier queue*. At the beginning of the algorithm the source is pushed into the queue.

The iterative step pops the top element of the frontier queue and propagates it to create potential windows. These potential windows are added to the corresponding edges to create new windows, which are pushed into the frontier queue. If the new windows are incident on a critical vertex, and the critical vertex is already present in the queue, it is updated with the new distance function. If the critical vertex is not in the queue, then it is pushed into the queue. If an existing window is modified due to its overlap with the potential window, then the queue is updated accordingly. This step is repeated until the queue is empty.

Note that a window popped off the frontier queue can be modified or deleted later by a potential window.

The final distance function obtained at the end of the algorithm is independent (except for small numerical inaccuracies) of the ordering of the queue. However the choice of this ordering function affects the time taken to compute the distance function over the whole mesh. For better performance we use the minimum value of the distance function over the window as the sorting criteria. For critical vertices, this value is the distance of the critical vertex from the source.

3.4 Adapting MMP algorithm to streaming setting

As in any Dijkstra based method that has an outward moving frontier region, to adapt the MMP algorithm to streaming setting, we first reorder the mesh in a spiral fashion around the source. The update step in MMP algorithm, however, is very different from that in Dijkstra's method or fast marching in the following respects:

- **Reordering:** The stream is reordered in a spiral fashion around the source vertex.

- **Propagation:** A window can be propagated only if the neighboring triangle is in memory. If a window is at the top of the frontier queue and the window's neighboring triangle is not in memory and the window does not lie on a corner edge, we read from the stream until we get the neighboring triangle or the edge turns out to be a corner edge. Then we propagate the window. For testing if an edge is a corner edge or not, we need to check if it has two adjacent triangles in the mesh or not. If one of the neighboring triangles of the edge is not in memory even after one of its neighboring vertices have been finalized, we know that the edge is a corner edge.

  A critical vertex can be propagated only when all of its neighboring triangles are in memory *i.e.* when the vertex has been finalized. If a critical vertex is at the top of the frontier queue and has not been finalized, we read from the stream until the vertex is finalized and then propagate it.

- **Updating existing windows:** As we have already seen, a window that has been popped off the frontier queue can be later modified by potential windows created by propagating windows occurring later in the queue. So we cannot discard a window from memory as soon as it is popped off the frontier queue. Let w_0 be a window which has been popped off the frontier queue, w_1 be a window which is at the top of the frontier queue at some time after the w_0 has been popped off. Let w_c be a potential window created by propagating w_1. We know that the minimum of the distance function on w_1 and thus that of w_c is not less than that of w_0. However it is possible that there is a point on which both w_c and w_0 are incident and the distance value at this point from w_c is lesser than that from w_0.

  If we keep w_0 in memory until the maximum distance value on w_0 is less than the minimum distance value on w_1 such a situation would not arise. So we discard a window from memory only if the maximum distance value on this window is less than the minimum distance value of the top element of the frontier queue. To implement this efficiently, we use another priority queue, called "inner frontier queue". This queue contains windows and is sorted on the maximum distance value on the window. A window which has been popped off the frontier queue is pushed into the inner frontier queue. At each iterative step of the algorithm, we pop off from inner frontier queue those windows on which the maximum distance has fallen below the minimum distance on the top element of the frontier queue. Once a window has been popped off the inner frontier queue, we delete it and check whether there are any windows left on the two neighboring triangles. If any of these triangles have no windows left, we delete those triangles and check if the finalized vertices of these triangles have any triangles left in memory. If not we delete these finalized vertices.

4 A-star: Single source, single destination shortest path algorithms

A-star is a widely used algorithm for computing shortest paths from a given source to a given destination. The A-star mesh traversal is similar to Dijkstra mesh traversal in that it uses a growing frontier to visit the mesh. However, the frontier in A-star is biased by a heuristic function to grow towards the destination. Thus in this case the frontier is elliptical in shape, compared to the circular frontier for Dijkstra-based algorithms.

Let s be the given source vertex and t be the given destination vertex and $d(u, v)$ be the distance from a vertex u to a vertex v as computed by Dijkstra's method . Let h be an A-star heuristic distance function defined on all the vertices of the mesh. The function h must satisfy

$$h(u) - h(v) \leq d(u, v) \quad (1)$$

$$h(t) = 0 \quad (2)$$

for all vertices u, v in the mesh

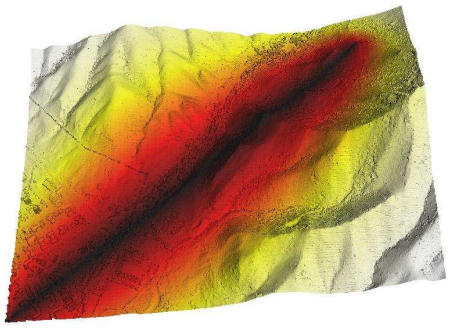


Figure 9: Streamed mesh reordered in an elliptical spiral around the source and target

We want $h(u)$ to be an approximation of $d(u,t)$ that satisfies the above two properties.

4.1 Adding A-star heuristic to Dijkstra's algorithm

Given an A-star heuristic function h, the A-star method is the same as Dijkstra's method except that the priority queue is ordered on $d(s,u) + h(u)$ for a vertex u, instead of $d(s,u)$. The algorithm is terminated when the destination t is popped off the queue.

We assume that an edge-length in our mesh is given by the Euclidean length of the edge. For such meshes, we use Euclidean distance from destination t as our A-star heuristic distance function for Dijkstra's method. Let $||\cdot||$ represent Euclidean distance. Then,

$$||u-t|| - ||v-t|| \leq ||u-v|| \leq d(u,v)$$

by the triangle inequality. And,

$$||t-t|| = 0$$

Thus Euclidean distance from t satisfies the properties 1 and 2 required of an A-star heuristic for Dijkstra's method.

We have seen that the frontier is elliptical for A-star based shortest path algorithms. Thus to adapt A-star based algorithms to streaming framework we reorder the stream into an elliptical spiral around the source and target vertex. See figure 9.

Results

On the reordered mesh (figure 9) the above algorithm took 206 Mb of memory and 13 seconds to run, while on the unordered mesh (figure 2(a)) it took 340 Mb of memory and 15 seconds to run.

4.2 Adding A-star heuristic to the MMP algorithm

The MMP algorithm computes a distance function that is defined on all points on all edges of the mesh. Using arguments similar to those for the Dijkstra's algorithm in the previous subsection, it can be seen that Euclidean distance to the destination t satisfies the properties required of an A-star heuristic for MMP method. Let $h(p)$ be the euclidean distance of the point p on an edge of the mesh from the destination t, and $d(s,p)$ be the distance from source s to p computed by the MMP algorithm.

We define two functions on the set of windows. For a window, w,

$$h_{max}(w) = \max_{p \in w} d(s,p) + \max_{q \in w} h(q)$$

and

$$h_{min}(w) = \min_{p \in w} d(s,p) + \min_{q \in w} h(q)$$

Here p and q are points in the interval corresponding to w.

The function h_{min} is also extended to the set of critical vertices. Let v be a critical vertex, then

$$h_{min}(v) = d(s,v) + h(v)$$

Let w be any window in the frontier queue and u a critical vertex in this queue. The following changes are made to the MMP algorithm to add A-star heuristic to it:

- The frontier queue for an unbiased MMP algorithm is ordered on $\min_{p \in w} d(s,p)$ for windows and $d(s,u)$ for critical vertices. For the A-star version we order this queue on $h_{min}(e)$, where e is a window or a critical vertex.

- The inner frontier queue for an unbiased MMP algorithm is ordered on $\max_{p \in w} d(s,p)$. For the A-star version we order it on $h_{max}(w)$.

- The top of the inner frontier queue, w_0, is popped off from the queue if and only if $h_{max}(w_0) \leq h_{min}(e)$, where e is the top element of the frontier queue. w_0 is then discarded from memory. The triangle and vertex deletion step is the same as in unbiased MMP algorithm.

- We terminate the algorithm when $d(s,t)$ is less than or equal to the minimum value of distance function on the top element of the frontier queue.

The termination condition given above ensures that the path from the source to the destination is indeed the shortest path. Note that ideally, we would like to use the following definitions for h_{max} and h_{min}

$$h_{max}(w) = \max_{p \in w}(d(s,p) + h(p))$$

$$h_{min}(w) = \min_{p \in w}(d(s,p) + h(p))$$

for a window w.

However, this function is much more difficult to compute than the one we have used, which serves as a close approximation.

Results

On the reordered mesh (figure 9) the A-star version of MMP algorithm took 470 Mb of memory and 120 seconds to run, while on the unordered mesh (figure 2(a)) it took 1 Gb of memory and 168 seconds to run.

5 Reordering

We have seen that reordering the mesh stream reduces the memory required for running shortest paths algorithms on the mesh. We use two different schemes for reordering. In the first one we project all points onto the x-y plane and then use the distance between the projected points and the projected source for reordering. The second scheme uses distance computed from Dijkstra's method for reordering.

5.1 Reordering by projecting points onto the x-y plane

In this scheme we project all the points in the stream onto the x-y plane and then compute the distance between the projected points and the projected source. We then bin all the points according to these distances. These bins are then concatenated in increasing order of distance from the projected source. We use this reordering scheme for input to Dijkstra's and fast marching method. We use this scheme as it requires a very small amount of knowledge for computing the distances between the points in the stream and the source, and thus requires a very small amount of memory.

For Dijkstra's method with A-star algorithm we bin the points in the order of the sum of the distances of the corresponding projected points from the projected source and the projected target.

5.2 Dijkstra-based Reordering

This reordering scheme is used for MMP method which has a much higher memory requirement than dijkstra's or fast marching method. Since the streaming version of MMP method has a very high memory requirement as well, we need a more sophisticated reordering scheme than the previous one for MMP method. We use dijkstra based reordering for this purpose. Given a mesh stream, we run the algorithm described in 3.1 on it. A vertex is output to the reordered stream, when it is popped of the priority queue. A triangle, all of whose vertices have been popped of the queue is output to the reordered stream. A vertex which has been popped off the queue and all of whose neighbors have also been popped off is finalized in the reordered stream. It can be seen that the reordered stream satisfies the two properties described in 2.

The reordering scheme for the MMP algorithm with A-star heuristic is very similar to the above reordering scheme. The two key differences are:

- Dijkstra's algorithm with A-star heuristic is run on the input stream.

- We reorder only that part of the mesh through which the shortest path between the source and the target can pass. Given an upperbound on the distance between the source and target and a lower bound for distance between any two points on the mesh we can compute if the shortest path can pass through a point or not. If the shortest path passes through a given point, then the sum of the lowerbound of the distance of that point from the source and lowerbound of the distance of that point to the target will be smaller than the upperbound of the distance from the source to the target. The Dijkstra distance between source and target is computed during the course of the Dijkstra's method with A-star heuristic. Euclidean distance is used as a lowerbound to distance between two points.

6 Future Work

Of the three algorithms we have discussed in this report, only two compute distances corresponding to shortest paths that can pass through the interior of triangles, namely the fast marching and MMP algorithms. The MMP algorithm computes exact distances, which can be used to trace back the shortest paths very easily. But it doesn't provide the flexibility to compute exact weighted shortest paths, as shortest paths in MMP are restricted to travel in a straight line. However we can use it to compute approximate shortest paths by allowing the shortest paths to bend when they cross an edge. The bending in this case is similar to the bending of a light ray due to refraction *i.e.* when the light ray crosses the boundary between two mediums of different optical density.

Fast Marching returns an approximate distance function and we have not yet found a stable way to compute shortest paths using this distance function. However, fast marching gives us a lot of flexibility since each triangle can have a different weight. But even with the flexibility of fast marching we cannot construct weights that penalize a path going uphill more than a path going down hill.

Notice that we have to do two passes through the mesh stream, once for reordering and once for shortest path computations. Ideally, we would prefer to eliminate the reordering pass since it does not fit well with the streaming ideology of reading a stream of points and processing them simultaneously. We have seen that the Dijkstra-based shortest path algorithms expect the vertices to appear in the stream in a particular kind of structure. Due to the non-local nature of the problem, it seems difficult for us to avoid the reordering step for a stream which does not have a structure favorable to shortest path computations. Say we have read a part of the stream and made some shortest path computations on the basis of that. These computations could easily become useless if a ridge near the source appears later in the stream.

For single source, single destination shortest path problems we have been experimenting with algorithms which use a lower bound (for distance between points on the mesh) and a suitable upper bound (for distance between source and target) to keep in memory only those parts of the mesh through which the shortest path can pass. Once we have read this part of the mesh into memory we can run standard shortest path algorithms on it. For the part of mesh kept in memory to be small, we need good upper and lower bounds. Without any previous knowledge of the mesh, efficiently computing such bounds would require reordering the stream. Currently we have been using Euclidean distance as a lower bound, as it does not need shortest path computations. However finding such an upper bound is still an issue.

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs.," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[2] R. Kimmel and J. Sethian, "Computing geodesic paths on manifolds," *Proc. of National Academy of Sci.*, vol. 95, no. 15, pp. 8431–8435, 1998.

[3] J. Mitchell, D. Mount, and C. Papadimitriou, "The discrete geodesic problem," *SIAM Journal of Computing*, vol. 16, no. 4, pp. 647–668, 1987.

[4] I. Pohl, "Bi-directional search," *Machine Intelligence*, vol. 6, pp. 127–140, 1971.