Fast Marching Minimal Path Extraction in ITK

Release 1.0

Dan Mueller[1]

March 4, 2008

[1]Queensland University of Technology, Brisbane, Australia

Abstract

This paper describes the ITK implementation of a minimal path extraction framework based on Fast Marching arrival functions. The method requires the user to provide three inputs: 1. a meaningful speed function to generate an arrival function, 2. path information in the form of start, end, and way-points (which the path must pass near), and 3. an optimizer which steps along the resultant arrival function perpendicular to the Fast Marching front. A number of perspectives for choosing speed functions and optimizers are given, as well as examples using synthetic and real images.

Keywords: *minimal path, centerline, vessel segmentation, ITK*

Contents

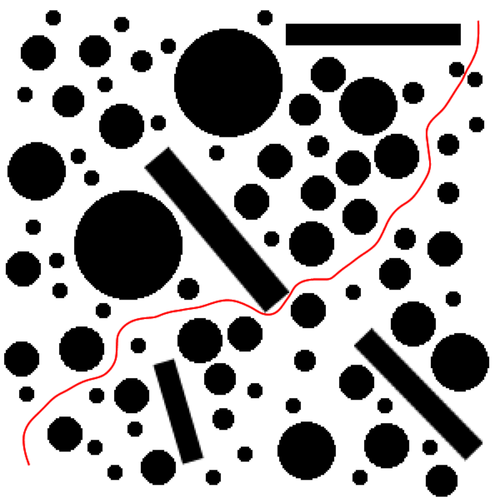
1 Background

Minimal path extraction is useful for a range of application domains including medical image analysis, robot navigation, and artificial intelligence (eg. emulating human behaviour in computer games). A number of optimisation approaches have been employed within these problem domains, such as tracking methods [3, 6], minimal cost methods (ie. Dijkstra's algorithm) [7], and A* (pronounced A star) [1].

Sethian [5, pp. 284-312] and Andrews et al. [2] explored the use of Fast Marching arrival functions to extract minimal paths. This method relies on the fact that the gradient of the Fast Marching arrival function T is orthogonal to the wave front. Furthermore, T has only one local minimum, with is guaranteed to be the global minimum (see [4, pp.17]). Therefore the minimal path can be extracted by back-propagating from a given seed (corresponding to the end point of the desired path) to the starting point implicitly embedded in T (see Figure 1). The back-propagation can be accomplished using the existing ITK optimizer framework (see Section 2.4).

The advantages of Fast Marching minimal path extraction are three-fold:

1. Fast Marching is computationally efficient with $O(n \log n)$ complexity.
2. The resultant path is continuous and directed (compare with skeletonization methods which return a set of unordered discrete points).
3. To the best of this author's knowledge, the generic method is not covered by any patents (however, please note that the framework *can* be used to implement vessel extraction methods which are covered by patents, as demonstrated in Section 2.3).


(a) Path

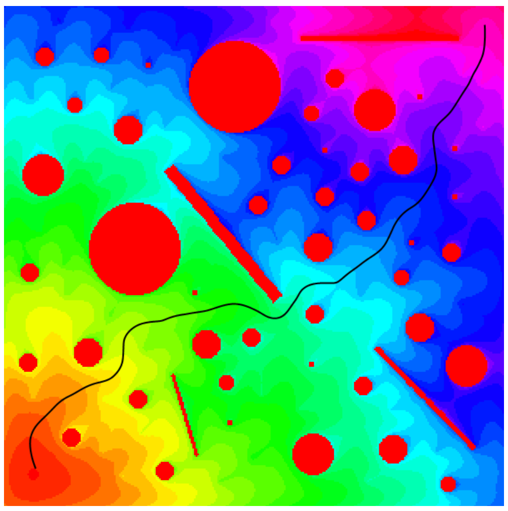

(b) Arrival

Figure 1: The minimal path is extracted by back-propagating from an end point to a start point embedded in the arrival function. Because the Fast Marching front moves orthogonal to itself, tracking perpendicular to the arrival function is guaranteed to follow the geodesic (minimal) path.

2 Implementation

2.1 Overview

This project implements a number of auxiliary functions, however the main filter expected to be employed by the user is `itk::SpeedFunctionToPathFilter`. This filter is a subclass of `itk::ArrivalFunctionToPathFilter`, which provides the basic functionality to convert an arrival function to a path. The main filter provides the added functionality of computing appropriate arrival functions from a given speed function, which are then feed to the subclass for processing. Figure 2 depicts an overview of `itk::SpeedFunctionToPathFilter`.

`itk::SpeedFunctionToPathFilter` expects path information (consisting of start, end, and optional way-points) and a speed function (a real-valued image in the range $[0, 1]$). Starting with the first way-point (or end point if there are no way-points) a front is propagated in the typical Fast Marching manner (each point from which a front is propagated is called a *Trial* point). The front terminates when both the previous and next points in the path have been reached (this is accomplished using the `itk::FastMarchingUpwindGradientImageFilter`). The resultant arrival function is viewed as a cost function using a new class called `itk::SingleImageCostFunction`, and is optimized using an appropriate `itk::SingleValuedNonLinearOptimizer`. The optimizer is initialised with the next point in the path list, and at each step the cost function (ie. arrival function) is minimised to step closer to the local (and global) minimum (ie. the *Trial* point).

`itk::SpeedFunctionToPathFilter` monitors the optimizer `Iteration` event, and saves each position to the current path. This means the filter only supports step-based optimizers, such as `itk::GradientDescentOptimizer`. Once the minimum of the current arrival function has been reached, the process is repeated with the next point in the list of path information. Once the path list has been exhausted, the process is repeated for each path (start, end, way-points) specified by the user. An extracted path is stored in a separate filter output (ie. specifying multiple path information objects generates multiple outputs).

At the moment paths are represented using `itk::PolyLineParametricPath`, which stores each point as an `itk::ContinuousIndex`. This situation is perhaps not ideal because the path points should really be stored in physical space (ie. using `itk::Points`). Additionally, the `itk::Path` framework does not support input/output operations (the user is required to implement reading and writing to files manually).

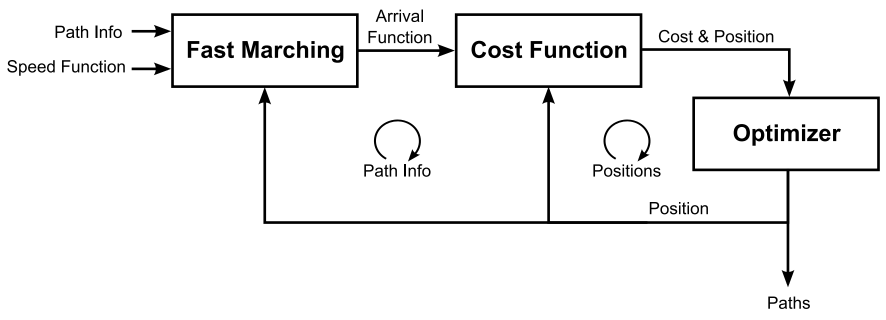


Figure 2: An overview of the internal workings of `itk::SpeedFunctionToPathFilter`.

2.2 Path Information

Fast Marching minimal path extraction is a semi-autonomous segmentation method — the user is required to provide start and end points. This is accomplished using an internal class `PathInfo` (see `itkSpeedFunctionToPathFilter.h`). The following code fragment demonstrates how the set the start and end points, and how to add a single way-point which the path must pass near.

```cpp
// Typedefs
const unsigned int Dimension = 2;
typedef float PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;
typedef itk::PolyLineParametricPath< Dimension > PathType;
typedef itk::SpeedFunctionToPathFilter< ImageType, PathType > PathFilterType;

// Setup path points
PathFilterType::PointType start, end, way1;
start[0] = 10; start[1] = 100;
end[0] = 100; end[1] = 10;
way1[0] = 10; way1[1] = 10;

// Add path information
PathFilterType::PathInfo info;
info.SetStartPoint( start );
info.SetEndPoint( end );
info.AddWayPoint( way1 );
pathFilter->AddPathInfo( info );
```

Multiple `PathInfo` objects can be added to the `itk::SpeedFunctionToPathFilter` instance using the `AddPathInfo()` method. Previously added path information objects added to the filter can be cleared using the `ClearPathInfo()` method.

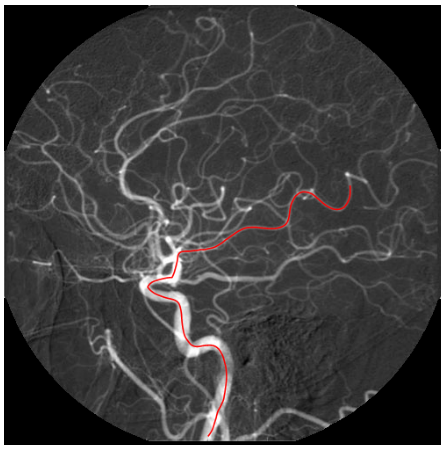
Because the user can not be expected to exactly specify points on the path, the `TerminationValue` parameter provides a configurable buffer which protects against the path needing to directly pass through each point. The optimizer is terminated when the current arrival value is less than `TerminationValue`; the smaller the value, the closer the path will get to each user specified point. The default value is 1.0 (note the parameter is specified in arrival time units, not physical spacing). To prevent oscillations, it is recommended that the specified optimizer has a small step size when `TerminationValue` is small.

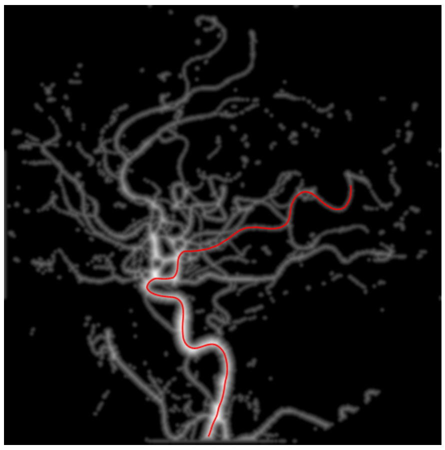
2.3 Speed Function

Choosing an appropriate speed function is the most difficult part of the entire process (similar to choosing a speed function when performing active contour segmentation). Recall that the speed function must be a real-valued image (ie. `float` or `double`) in the range $[0,1]$. For computational efficiency it is desirable to have the speed function as close to 1.0 near the desired path, and 0.0 elsewhere. In some situations this may not be possible, in which case there will be a computational penalty because the Fast Marching front will visit most (if not all) of the speed function.

TODO: Discuss efficient method for generating speed function for vessel segmentation.

When using the framework to extract the medial axis of a vessel, it may be necessary to use a method to center the path. Deschamps [4] proposed one such method which firstly used the method to extract a rough centerline, performed segmentation using the rough centerline, then computed a speed function using the signed distance transform of the resultant binary volume, and repeated the minimal path extraction. This method is depicted in Figure 3. It should be noted that this method (ie. extracting centered paths using a distance transform) may be covered by a patent. As can be seen, because the minimal path extraction framework described in this paper is capable of implementing a range of methods, care must be taken to ensure the framework is not used to implement a patented method.


(a) Uncentered Speed


(b) Centered Speed

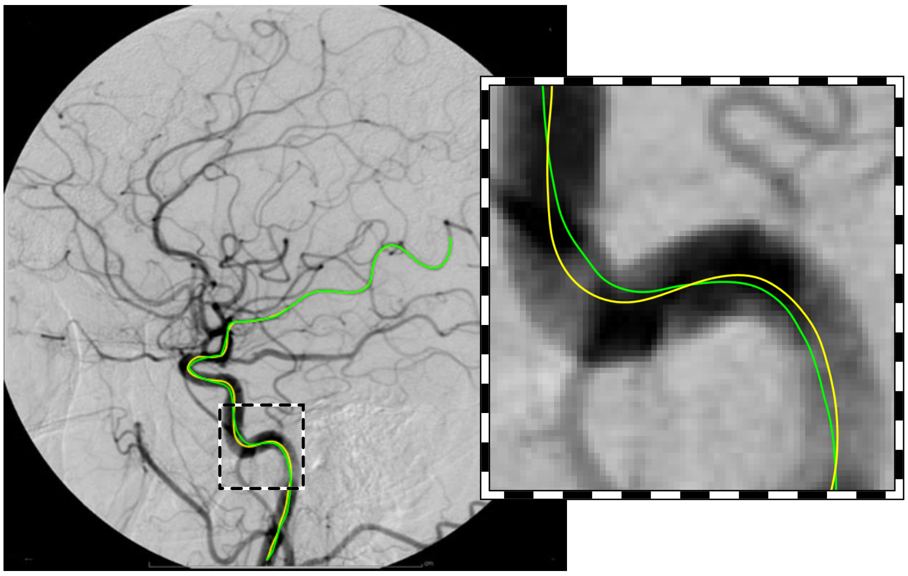
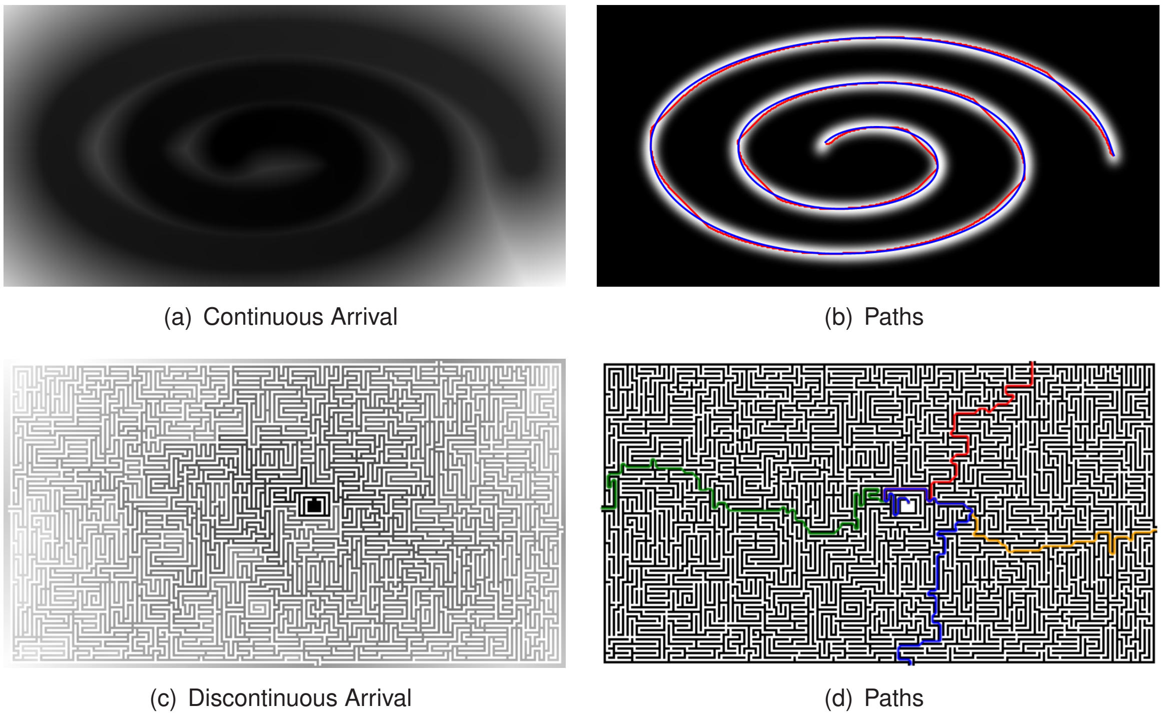

(c) Overlay

Figure 3: A digital subtraction angiography (DSA) image of the cerebral vessels. The resultant paths have been laid over the original image: the minimal path (green) and centered path (yellow). Original image courtesy of Wikipedia.

2.4 Optimizer

At present there are three step-wise optimizers which function together with the given minimal path framework: `GradientDescentOptimizer`, `RegularStepGradientDescentOptimizer`, and `IterateNeighborhoodOptimizer`. `IterateNeighborhoodOptimizer` is a new optimizer which operates by iterating the neighbors of the current position, and moving to the minimum value.

The gradient descent methods are suitable for minimizing continuous cost functions (ie. functions without discontinuities). In comparison, the neighborhood method is suitable for minimizing cost functions with large discontinuities (ie. when pixels visited by the arrival function are directly adjacent to non-visited pixels). The gradient descent optimizers are not suitable for discontinuous cost functions because the step vector, which is computed using the gradient, is skewed in regions direct adjacently to non-visited pixels. Examples of continuous and discontinuous cost functions are depicted in Figure 4. It is anticipated that for most medical imaging applications — in which the speed function is continuous — the gradient descent optimizers will be most suitable.


(a) Continuous Arrival

(b) Paths

(c) Discontinuous Arrival

(d) Paths

Figure 4: The top images (a and b) depict a continuous arrival function and extracted paths using the gradient descent (blue) and iterate neighborhood (red) optimizers for a synthetic dataset (`Data/Synthetic-02`). The bottom image (c) depicts a discontinuous arrival function (note that visited pixels are directly adjacent to non-visited pixels). The maze image (`Data/Synthetic-03`) is courtesy of Wikipedia.

3 Examples

The code from these examples can be found in `Source/examples.cxx`.

3.1 Using the Gradient Descent Optimizer

The first step for using the proposed framework is to setup the relevant type definitions:

```
const unsigned int Dimension = 2;
typedef float PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;
typedef itk::PolyLineParametricPath< Dimension > PathType;
typedef itk::SpeedFunctionToPathFilter< ImageType, PathType > PathFilterType;
```

Next create a cost function:

```
PathFilterType::CostFunctionType::Pointer cost =
    PathFilterType::CostFunctionType::New();
```

Now we create and setup the desired optimizer, noting that the `LearningRate` parameter will default to `1.0`:

```
typedef itk::GradientDescentOptimizer OptimizerType;
OptimizerType::Pointer optimizer = OptimizerType::New();
optimizer->SetNumberOfIterations( 1000 );
```

Finally, we plug the cost function and optimizer into a newly created path filter object:

```
PathFilterType::Pointer pathFilter = PathFilterType::New();
pathFilter->SetInput( speed );
pathFilter->SetCostFunction( cost );
pathFilter->SetOptimizer( optimizer );
pathFilter->SetTerminationValue( 2.0 );
```

The path information is specified as described in Section 2.2 and the filter updated to compute the path.

3.2 Using the Regular Step Gradient Descent Optimizer

The regular step gradient descent optimizer gives the user more control over the step size, via the `SetMaximumStepLength`, `SetMinimumStepLength`, and `SetRelaxationFactor` methods. In the following example, the step length is configured to be roughly `0.5` but can vary as small as `0.1`:

```
typedef itk::RegularStepGradientDescentOptimizer OptimizerType;
OptimizerType::Pointer optimizer = OptimizerType::New();
optimizer->SetNumberOfIterations( 1000 );
optimizer->SetMaximumStepLength( 0.5 );
optimizer->SetMinimumStepLength( 0.1 );
optimizer->SetRelaxationFactor( 0.5 );
```

4 Conclusion

This paper has described the implementation of Fast Marching minimal path extraction for ITK. The framework allows users to specify a speed function, from which a Fast Marching arrival function is computed and the geodesic (minimal) path between a number of points is extracted. The proposed implementation uses the existing optimizer framework within ITK to perform the minimal path extraction. For suggestions or bugs, feel free to contact the author[1].

[1]Corresponding author: Dan Mueller: d.mueller@qut.edu.au or dan.muel@gmail.com.

References

[1] ________. A* search algorithm. Technical report, Wikipedia, the free encyclopedia, 2007, Available online: http://en.wikipedia.org/wiki/A*_search 1

[2] J. Andrews and J. Sethian. Fast marching methods for the continuous traveling salesman problem. *Proceedings of the National Academy of Sciences (PNAS)*, 104(4):1118–1123, 2007. 1

[3] S. Aylward and E. Bullitt. Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction. *IEEE Transactions on Medical Imaging*, 21(2):61–75, 2002. 1

[4] Thomas Deschamps. *Curve and Shape Extraction with Minimal Path and Level-Sets techniques: Applications to 3D Medical Imaging*. PhD dissertation, University of Paris Dauphine, 2001, Available online: http://math.lbl.gov/~deschamp/html/phdthesis.html 1, 2.3

[5] J. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge Press, 2^{nd} edition, 1999. 1

[6] O. Wink, W. Niessen, and M. Viergever. Fast delineation and visualization of vessels in 3-D angiographic images. *IEEE Transactions on Medical Imaging*, 19(4):337–346, 2000. 1

[7] O. Wink, W. Niessen, and M. Viergever. Multiscale vessel tracking. *IEEE Transactions on Medical Imaging*, 23(1):130–133, 2004. 1