SANDIA REPORT

SAND2016-4557
Unlimited Release
Printed September 2015

Path Network Recovery Using Remote Sensing Data and Geospatial-Temporal Semantic Graphs

William C. McLendon III and Randy C. Brost

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2016-4557
Unlimited Release
Printed September 2015

Path Network Recovery Using Remote Sensing Data and Geospatial-Temporal Semantic Graphs

William C. McLendon III
Data Driven & Neural Computing
Sandia National Laboratories
Albuquerque, New Mexico 87185-1326
wcmclen@sandia.gov

Randy C. Brost
Data Driven & Neural Computing
Sandia National Laboratories
Albuquerque, New Mexico 87185-1326
rcbrost@sandia.gov

Abstract

Remote sensing systems produce large volumes of high-resolution images that are difficult to search. The GeoGraphy (pronounced Geo-Graph-y) framework [2, 20] encodes remote sensing imagery into a geospatial-temporal semantic graph representation to enable high level semantic searches to be performed. Typically scene objects such as buildings and trees tend to be shaped like blocks with few holes, but other shapes generated from path networks tend to have a large number of holes and can span a large geographic region due to their connectedness. For example, we have a dataset covering the city of Philadelphia in which there is a single road network node spanning a 6 mile $\times$ 8 mile region. Even a simple question such as "find two houses near the same street" might give unexpected results.

More generally, nodes arising from networks of paths (roads, sidewalks, trails, etc.) require additional processing to make them useful for searches in GeoGraphy. We have assigned the term *Path Network Recovery* to this process. Path Network Recovery is a three-step process involving (1) partitioning the network node into segments, (2) repairing broken path segments interrupted by occlusions or sensor noise, and (3) adding path-aware search semantics into GeoQuestions.

This report covers the path network recovery process, how it is used, and some example use cases of the current capabilities.

Acknowledgments

We gratefully thank Jarlath O'Neil-Dunne and his group at the University of Vermont for their excellent work preparing the raw data and land cover maps used in our experiments.

This work was funded by the Sandia National Laboratories LDRD program as a part of the Pattern Analytics to Support High-Performance Exploitation and Reasoning (PANTHER) grand challenge project and also by the DOE/NNSA Office of Nonproliferation Research and Development (NA-22). Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

Nomenclature 9

1 Introduction 11

2 Path Partitioning 13
- 2.1 Path Partitioning Background 13
- 2.2 Technical Details 15
- 2.3 Path Partitioning Usage 24

3 Path Connected Search 33
- 3.1 Path Connected Search Background 33
- 3.2 Path Connected Search Technical Details 34
- 3.3 Path Connected Search Usage 37

4 Discussion 46

Appendix

A Overview of the "Driving Home" Search 48

B Match Hierarchy 52

References 54

Figures

1 Example: Road Network Shape 13
2 Path Partitioning Process 14
3 Example: Skeleton from a Voronoi Diagram 16
4 Example: Euclidean Distance Transform 17
5 Algorithm: Zhang-Suen Skeletonization by Thinning 18
6 Example: Shape Skeleton 19
7 Algorithm: Identify Critical Points 20
8 Algorithm: Identify Critical Points 4-way 21
9 Example: Partitioned Shape after Fill 22
10 Path Partitioning StoredGraph Result Structure 23
11 Driving Home Example: Land cover image of scene 24
12 Driving Home Example: Road network 32
13 Driving Home Example: Partitioned Road Segment (graph only) 33
14 Driving Home Example: Road Segments with Land Cover 34
15 Path-Connected Edge Diagram 36
16 Driving Home Example: Shortest Path 45
17 Driving Home Example: Shortest Path (graph only) 46
18 Driving Home Example: Path-Connected Edge 47
19 Driving Home Example: Path-Connected Edge (graph only) 48

Nomenclature

AddToStoredGraph	An application of the GeoGraphy library that constructs a StoredGraph from input data files.
GeoGraphy	Pronounced ”Geo-Graph-y”. This is the C++ code framework for our geospatial-temporal semantic graphs.
GeoQuestion	A query executed against the graph that specifies patterns and/or geometric operations to be performed on the nodes. GeoQuestions are constructed from seven input files: {SearchDefinition, Role, NodeSpecList, EdgeSpecList, RoleNode, RoleEdge, and PostprocessDefinition}.
GeoSearch	GeoSearch is an application that comes with the GeoGraphy package. This application is the primary means to execute search queries against a StoredGraph. It takes a *GeoQuestion* and a *StoredGraph* as inputs and will return any results found.
GIS	**G**eographic **I**nformation **S**ystem is a system that stores, analyzes, and manipulates spatial and geographical data for visualization.
GSTSG	Geo-Spatial Temporal Semantic Graph.
Land cover	A posterized representation in a geospatial image format (such as a GeoTiff or ESRI Image file) in which pixels are assigned a value based on the classification of what is at that location. Typical classes might include Buildings, Road, Railroad, Water, Tree Canopy, etc.
LiDAR	**Li**ght **D**etection **A**nd **R**anging is a remote sensing technology that generates 3D maps by illuminating regions using a laser to measure distances from the sensor to objects.
QGIS	The Quantum Geographic Information System (QGIS) [19] is an open source GIS software package based on the GDAL framework. The GeoGraphy team makes extensive use of this tool for our visualization needs.
SearchGraph	A temporary graph constructed by the GeoSearch application to fulfill a GeoQuestion. SearchGraph nodes are assigned roles corresponding to those specified in the GeoQuestion.
StoredGraph	A persistent graph, containing the history of image and other data that has been input. Currently it is saved in a SQLite database.

Introduction

Collection platforms ranging from UAVs to satellites are able to obtain a vast quantity of imagery. Unfortunately, progress in analysis techniques to process this data has not kept pace with increased collection rates, since many analysis activities remain a largely manual task. We have developed a *geospatial-temporal semantic graph* (GSTSG) toolkit, GeoGraphy, that constructs a GSTSG from imagery products that can then be searched. By moving our search into an object-based approach rather than pixel-based we can ask questions more easily about the structure of items that we might be interested in.

In the GeoGraphy pipeline, we process one or more posterized images called land cover images using AddToStoredGraph into a GSTSG that we call the StoredGraph. The StoredGraph at this time is a SQLite database for the convenience of having a file-based dataset. An example of a land cover image can be found in Figure 11. They are currently generated using the technique developed by Jarlath O'Neil-Dunne at the University of Vermont [17, 15, 18, 16] by combining imagery data, elevation data (LiDAR), and GIS data (road centerlines or polygons) using eCognition [8]. The resulting posterized image can be thought of as a "paint by numbers" image in which every pixel from the original scene is assigned a color that corresponds to a land cover class of what is located on that spot. Land cover classes include *Building*, *Roads*, *Grass/Shrub*, *Other Paved*, *Water*, *Tree Canopy*, *Bare Earth*, *Railroad*, etc. Graphs are constructed by scanning the land cover image and generating a graph node for each connected region of each color class. For many objects such as buildings the nodes generated are well behaved, that is they are reasonably well defined where each node represents an individual building or contiguous structure in the case of rows of townhouses in urban environments. In other cases the graph nodes are more difficult to work with because there are large regions of pixels containing the same class such as forests, bodies of water, road networks, etc. Our motivation stems from the road network example. Additional information regarding the use of GeoGraphy for activity based analysis can be found in [3].

Unfortunately, some nodes such as road networks are difficult to deal with because they can cover large geospatial regions with a large number of holes in them and be highly connected to a large portion of our graph. This makes useful semantic searches difficult because a single "road" node might be proximate to a large portion of the graph. This does not match how a human would typically describe a road—we tend to think of roads in smaller units such as "blocks." A human generally does not think of the residential street in front of their house as being part of the interstate highway system, even if it is essentially part of a contiguous piece of pavement that connects the whole continent. Further, if we wish to reason about the area or relative direction to a path from some other node such as a building, we must break these large road networks into more manageable pieces. Therefore, if we wish to incorporate network style nodes into GeoGraphy in a way that is useful for search operations, we must perform further processing on these nodes. We call this operation *path network recovery*.

The general path network recovery problem is organized into three search steps. The first step is *path partitioning*, in which we apply morphology operations to break the network node into segments approximately at intersections. We use the write-back feature of GeoGraphy to save the results of the partitioning search back to the StoredGraph. The second step is *path repair*, in which broken path segments are connected to overcome degradation due to occlusions or sensor noise. These results are also written back to the StoredGraph. There are multiple approaches to path repair, sometimes applied before path partitioning. The third step, called *path connected search*, uses the resulting path segments and connections to search the topology of the path network to generate path-connected edges that represent a new type of edge connecting nodes. These new edges are then used in the *Star* and *Interrupted Star* graph searches to answer questions that include path-finding, reachability, etc.

The remainder of this document will follow a “Driving Home” scenario, in which we will cover the necessary GeoQuestion elements to answer the question “What route along a road network might someone follow to get from a parking lot to their house.” This scenario will demonstrate steps one and three, path partitioning and path-connected search.

Two additional examples are described in Appendices C and D, published in a separate Official Use Only SAND report[13]. The final example given in Appendix D will demonstrate steps one and two, path partitioning and an approach to path repair.

Path Partitioning

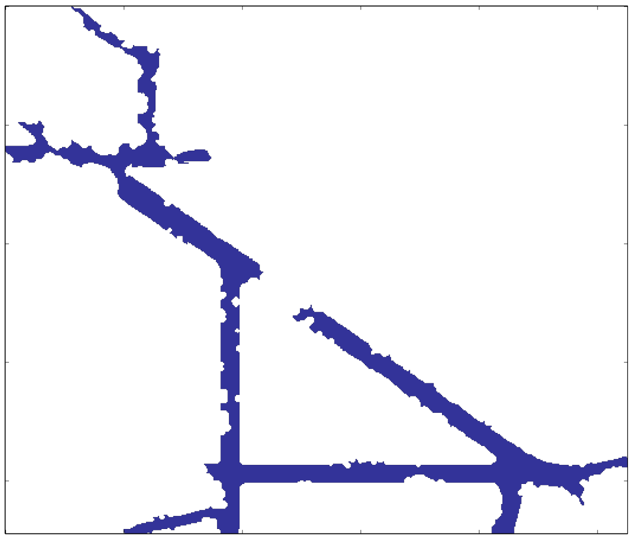


Figure 1. The shape of a region node of a section of road network. The various notches taken out result from tree canopy overhanging the road.

Path Partitioning Background

Path partitioning could possibly mean many things, depending upon the context in which it is used. For our purposes, the stated goal of the path partitioning problem is this:

> *Partition a region node representing a complex network of intersecting paths into individual segments by breaking it at intersections, then further partition any long segments exceeding a maximum length threshold.*

To accomplish this, we investigated several techniques to identify the method that might work best given the following constraints within our environment:

- We do not have vector representations of our shapes.
- Region node shapes are stored in a Run Length Encoded (RLE) data structure.
- Segment widths vary between intersections, possibly with large "bites" removed due to tree canopy or other overhanging obstructions.
- Partitioning at intersections is preferred.
- Preference is given to techniques that are more straightforward to implement.

Figure 1 shows an example region node from a road network. In this case the segment was extracted in a wooded area that had a lot of tree canopy overhanging the road in many places. These overhangs complicate the shape geometry of the road by removing "bites" from the road in various places. These can trick a partitioning algorithm into thinking an intersection has occurred at that location, a condition we would prefer to minimize.

A major driver early in the development process was the desire to partition our shapes roughly at intersections. To accomplish this we chose an approach that first extracts the shape skeleton. A *skeleton* is a thin-line representation of an object which provides a useful encoding that encapsulates both boundary and region information. We chose to generate a skeleton from our shapes because we can use it to easily identify the critical points of interest in our shape, specifically the *intersections* and *endpoints*. Figure 2 illustrates the overall process that we use for our shape partitioning code in the path partitioning pipeline. We will provide details of each step in the Technical Details section.

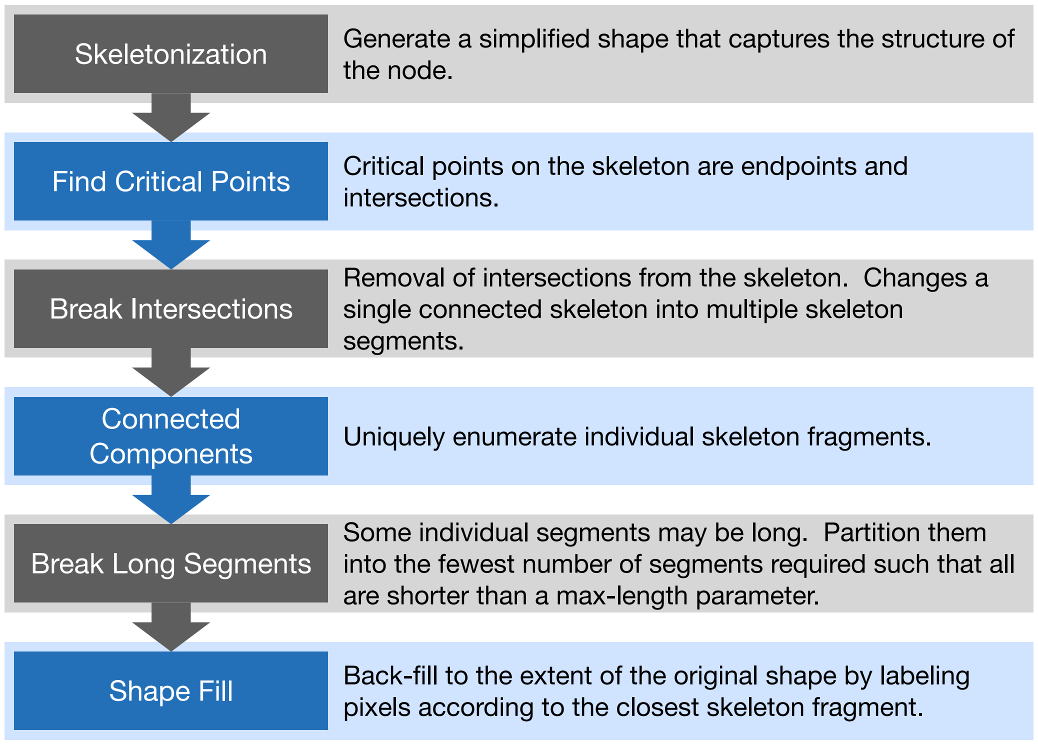

Skeletonization
Generate a simplified shape that captures the structure of the node.
Find Critical Points
Critical points on the skeleton are endpoints and intersections.
Break Intersections
Removal of intersections from the skeleton. Changes a single connected skeleton into multiple skeleton segments.
Connected Components
Uniquely enumerate individual skeleton fragments.
Break Long Segments
Some individual segments may be long. Partition them into the fewest number of segments required such that all are shorter than a max-length parameter.
Shape Fill
Back-fill to the extent of the original shape by labeling pixels according to the closest skeleton fragment.

Figure 2. The general process pipeline for the path partitioning process phase of path network recovery.

The ultimate goal of this activity is to provide search capabilities that are aware of the connectivity between structures in the ground model. Prior to this work, we relied on simple proximity metrics to determine if two facilities are "close" to another in terms of straight-line distance. Since actual ground-based objects do not move along the ground in straight-line paths, we needed to develop a way to ask questions that can use path-connectivity as a part of the query semantics.

Technical Details

In this section we will delve into the details of each of the main steps in the path partitioning pipeline as shown in Figure 2.

Skeletonization

Generating a thin line representation of a complex geometry, or skeletonization, has been well studied. There are numerous techniques available that include Medial Axis Transforms (MAT) [9], using voronoi diagrams [9], a Euclidean Distance Transform (EDT) [6, 14, 4], and thinning algorithms [1, 21, 12]. In GeoGraphy we do not store a vectorized representation of our shapes because generation of polygons from raster data can be time consuming. Instead we use a Run Length Encoding (RLE) method to store the raster data efficiently. While RLE is space-efficient, it does not provide random access to individual pixels. That is, you cannot count on accessing a pixel by its X and Y coordinate directly in O(1) time.

Medial Axis Transforms are an obvious choice to consider to construct the skeleton, but their implementations can be expensive to compute. Since GeoGraphy is designed to address wide-area search, which can include hundreds of thousands or millions of nodes, we have a strong preference for fast techniques. Because of this, we considered other possibilities.

Another method to find a skeleton is to compute the Voronoi diagram and Delaunay triangulation of the shape perimeter. Voronoi diagrams partition a space into cells in which each cell contains exactly one generating point and all points that are nearer to this generating point than to other generating points. As we increase the number of generating points, certain interior cell borders will map to the skeleton. Voronoi diagrams can be computed efficiently using Fortune's Algorithm [11] in $O(n \log n)$ time. Fortune's algorithm is a sweep-line method which would work well with our RLE encoded pixels, but the implementation is nontrivial and if we lack high resolution the quality of the skeleton may be low. Figure 3 shows an example of a running track region around a football field in which perimeter pixels are used as points to generate a Voronoi diagram. One thing that is obvious is the noise in the Voronoi diagram–choosing which boundaries to use as our skeleton can become more difficult with complex/irregular shapes.

The next method we investigated was a pure pixel-based method to compute the EDT of our shapes. Chow et. al. demonstrated that the EDT can be used to generate a skeleton [4] by identifying ridges. We performed some tests and found that finding high quality skeletons using this method for our shapes was challenging due to the complexity of our shapes. Figure 4 shows our roadway example with an EDT applied.

Finally, we investigated *thinning algorithms* for our skeleton. These have the advantage of being straightforward to implement, can work in raster space easily, and they produced skeletons that were both thin and had relatively few spurious branches. We implemented two algorithms [21, 12] that are derivatives of the one described by Hilditch [1], and ultimately settled on the Zhang-Suen variant for use in GeoGraphy.

The Zhang-Suen algorithm is fairly straightforward in its implementation, requiring roughly 30

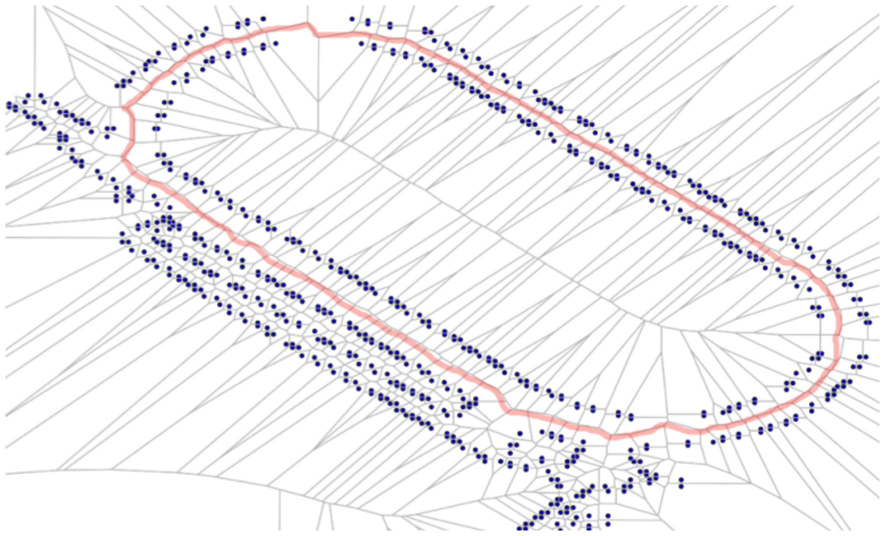


Figure 3. An example of a Voronoi diagram generated using points on the running track surrounding a football field. The 'skeleton' boundaries are highlighted.

lines of code to implement. It is also easily parallelizable if needed in the future. The pseudocode is shown in Figure 5.

The skeletonization code works on a binary raster representation of the object by extracting a 2D pixel array from the RLE shape. Background pixels are given the value of 0 and shape pixels a value of 1. The code is fast and parallelizable and the running time is proportional to the maximum *thickness* of the shape which dictates the number of iterations in the outer loop (line 01). In our cases, this should generally be small since we wish to apply the transformation to path-network style shapes.

The Zhang-Suen thinning algorithm operates by deciding whether or not a given non-background pixel, p_1, should be deleted (i.e., changed to a background pixel). To do so it looks at the set of 8 neighboring pixels to make the decision. A diagram showing the adjacent pixel numbering is shown in Figure 5. We won't go into details regarding the calculation of all of the parameters *A-D* in the algorithm, but the calculation of *A* is noteworthy because we adapt the technique for our *Identify Critical Points* step, described in Section 2.2. The calculation of **A** is performed by counting the number of zero to one transitions as we walk around the 3 x 3 grid from p_2 to p_3, p_3 to p_4, p_4 to p_5, ..., and p_9 to p_2. Effectively, this number counts the number of non-touching groups of background pixels adjacent to p_1. To remove the pixel, the adjacent background pixels must all be grouped together. Later, during classification, we will use this to determine if a skeleton pixel should be classified as an *intersection*, an *endpoint*, or a *skeleton* pixel.

Figure 6 shows the resulting skeleton after the thinning algorithm has been applied to our test shape.

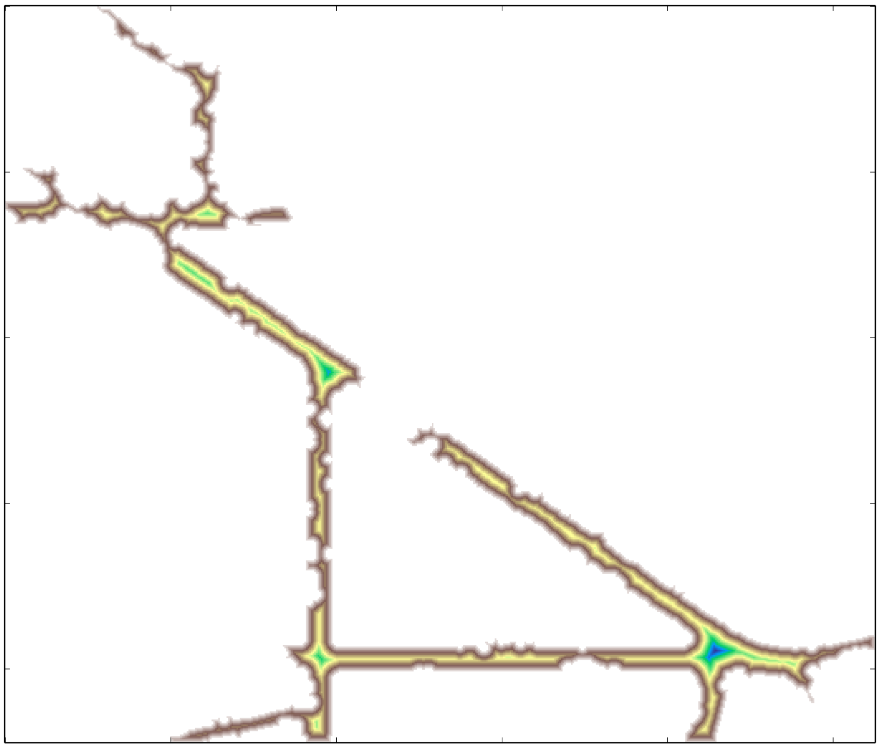


Figure 4. The road shape with an EDT applied. Ridges are plainly visible in the map, but the variations in magnitude and non-homogeny of width of the shape make it challenging to find the EDT in some parts of the shape.

Identify Critical Points

The overall goal of the path partitioning phase of the *path network recovery* operation is to break the shapes roughly at their intersections. To accomplish this we computed the skeleton, which we will now use to identify *critical points*. There are three types of critical points on the skeleton:

Intersection	A pixel is labeled as an intersection if it connects three or more paths.
End Point	A pixel is an end point if there are pixels leaving on only one side from it.
Skeleton Pixel	A pixel is a skeleton pixel if it connects two paths.

We can use a modification of the calculation of *A* in the Zhang Suen thinning algorithm to identify our critical points by observing that walking around the perimeter of p_1 is counting the number of separate components of our skeleton that are adjacent to p_1. We can think of each of these components as a separate path leaving p_1, so we can assign roles to the the pixels by simply counting the number of 0 to 1 transitions in the adjacent pixels. In some cases, it can be possible that a skeleton could consist of a single pixel. In this case, p_1 will have no neighboring non-background pixels, which means there will be no 0-to-1 transitions to count. This case is easily handled by checking for zero and then classifying these pixels as *end points*. Figure 7 shows the calculation to identify the critical points in our skeleton. The observation that the calculation of

Algorithm: Zhang-Suen Thinning

```
WHILE points are deleted DO:
    FOR each pixel p_1 in p(i,j) DO:
        A = # of 0 to 1 transitions in a clockwise direction around p(i,j)
        B = # of nonzero neighbors of p(i,j)
        C = p_2 x p_4 x p_6        (on odd iterations)
            p_2 x p_4 x p_8        (on even iterations)
        D = p_4 x p_6 x p_8        (on odd iterations)
            p_2 x p_6 x p_8        (on even iterations)
        IF A == 1 AND (2 <= B <= 6) AND C==0 AND D==0:
            Delete pixel p_1
        ENDIF
    ENDFOR
ENDWHILE
```

p_9	p_2	p_3
p_8	p_1	p_4
p_7	p_6	p_5

pixel ordering

Figure 5. The Zhang-Suen thinning algorithm [21] computes a pixel based skeleton of a binary raster image using an iterative thinning approach by deleting a pixel, p_1, if a set of conditions on its neighboring pixels is met.

A from the skeletonization code effectively calculates the number of skeleton pathways that leave pixel p_1 gives us a convenient metric to identify our critical points. This operation also has the desired quality that it can be trivially parallelized for multi-core or GPU applications.

There is one special case that we must consider in our critical point identification code. In some cases the intersection is defined by a grouping of four pixels rather than just one. This typically happens when four or more paths intersect, but it may be possible even when three paths intersect, we have not performed an exhaustive analysis of what conditions will cause the thinning algorithm to produce this construction. We identified this case through observations of the performance of the code on real data. Figure 8 shows a straightforward extension to find these special case intersection pixels by utilizing a 4x4 grid. Pseudocode for this is shown in Figure 8. Since we are only searching for instances of these special case intersection pixels, this operation is fast and runs in worst case $O(r * c)$ time where r is the number of rows and c is the number of columns of pixels in the raster image.

We should note that speed improvements on the skeletonization and critical points algorithms can be achieved in implementation if we keep track of only the non-zero pixels in the raster, and may reduce the execution time of these operations when the regions we are partitioning are especially sparse. After the critical points have been identified, the next step is to break the skeleton apart at the intersections.

Break Skeleton at Intersections

Breaking the skeleton apart at the intersections is done because we eventually would like to generate individual shapes for each of the segments of the original path shape. To break the skeleton, we simply change all intersection pixels as well as their eight neighboring pixels into background pixels. We change the neighboring pixels as well as the intersection pixel to ensure that we are able to break up the skeleton cleanly. This also cleans up the overall result by preventing over-segmentation due to stitching that can occur when very thin paths intersect at very shallow angles. After partitioning the skeleton, we recompute the critical points, which at this time should consist

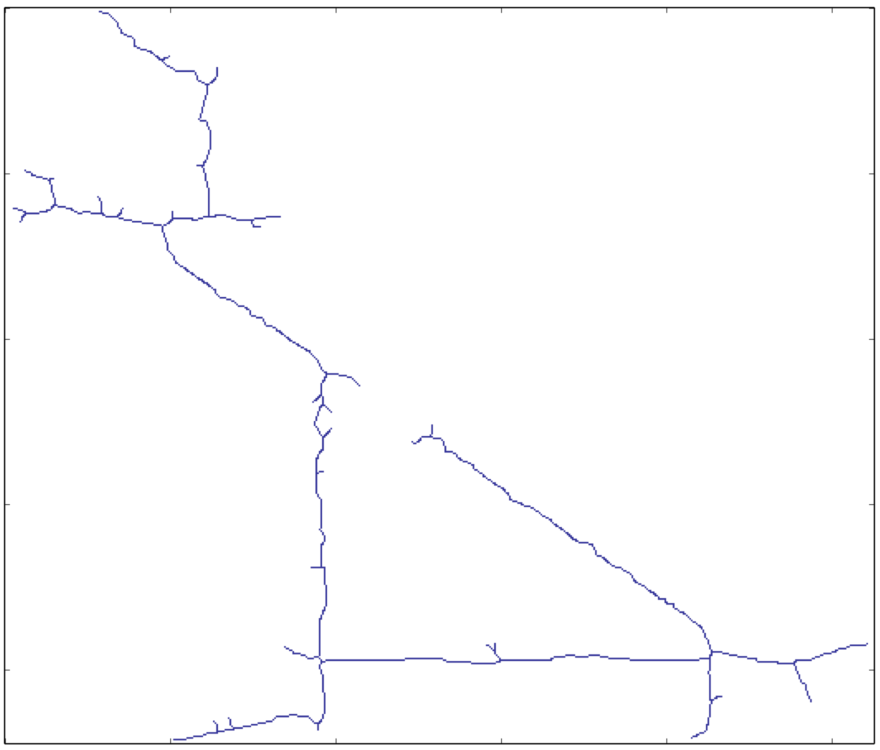


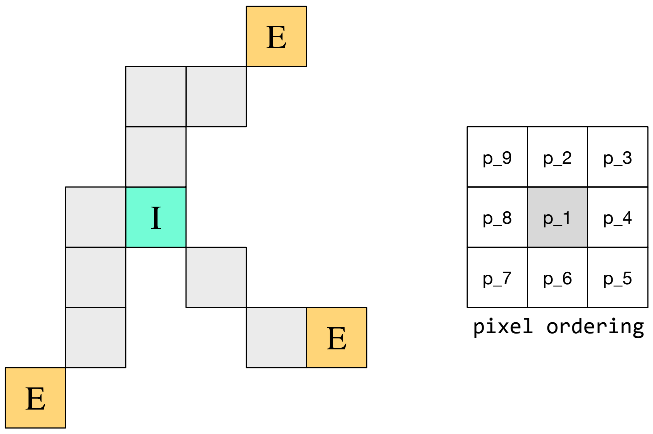
Figure 6. The resulting shape after the Zhang-Suen thinning algorithm was applied, resulting in a thin skeleton of the original shape.

of only skeleton nodes and endpoints, so we do not need to execute the special case check shown in Figure 8. We also note here that this step is likely to generate numerous singleton pixels, but they will be classified into end points properly by the pixel classification algorithm. We will use the end point pixels in the next step to group the pixels into enumerated segments.

Identify Skeleton Connected Components

The next step is to identify the connected components of the skeletons. This is performed by scanning the raster to produce a list of endpoint pixels. Next we select one of these pixels as a starting point and label all the pixels that are connected to it as a connected component. We determine connectivity in this step by performing a brushfire traversal of connected pixels starting from each endpoint. In this step we consider two pixels $p(r_1,c_1)$ and $p(r_2,c_2)$ to be connected if $p(r_2,c_2)$ is one of the eight neighbors of $p(r_1,c_1)$. Individual skeleton pixels are set to the index of their respective connected components in this step and are numbered from 1 to n.

At the end of this step, we apply the user threshold constraint on the size of connected components to eliminate components that are "too small." More discussion on this parameter will be provided later when we cover the GeoQuestion specification for partitioning problems. The connected components that remain are the path segments taken into the next phase of the computation.



```
Algorithm: Critical Points (1)
FOR each pixel p_1 in p(i,j) DO:
    A = # of 0 to 1 transitions in a clockwise direction around p(i,j)
    IF A == 0 THEN p_1 is an end-point pixel
    IF A == 1 THEN p_1 is an end-point pixel
    IF A == 2 THEN p_1 is a skeleton pixel
    IF A >= 3 THEN p_1 is an intersection pixel
ENDFOR
```

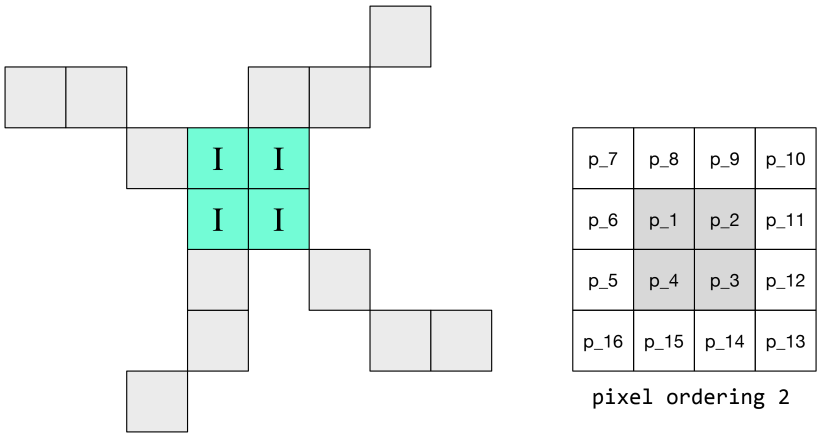
Figure 7. Algorithm that identifies critical points in the skeleton.

Break Long Skeleton Segments

In some cases, there can be long path segments that are not broken by an intersection. For some problems it may be desirable to allow this segment to be be retained as a singular segment, but we would like to allow the user the freedom to further partition our segments into smaller pieces. This allows us to further clean up particularly long segments that might still be difficult to deal with, such as very long or circular paths that have no intersections but would still be difficult to work with. This also results in a partitioning in which the number of "hops" along a path is loosely proportionate to the length of that path. This will be useful in the SimpleST path search, which will be discussed later in this document. Lastly, by breaking long pieces into smaller pieces, the relative direction between path segments and other scene objects can be more meaningfully computed, since the GeoGraphy code computes relative direction as the angle of the line connecting object centroids.

Shape Fill

The last step in the partitioning process is to recover the original shape as a collection of individual segments. This action is performed using two rasters as input: one containing the segmented skeleton, and the other containing the original binary image raster. The segmented skeleton pixels will contain values equivalent to the segment's connected component number and will be labeled from 1 to *n*. We perform a breadth-first expansion from all of the labeled pixels in the skeleton



```
Algorithm: Find 4-way intersections
FOR each pixel p_1 in p(i,j) DO:
    IF p_1 AND p_2 AND p_3 AND p_4 are not background pixels THEN
        A = # of 0 to 1 transitions in a clockwise direction around
            the outer pixels (i.e., p_5, p_6, p_7, … , p_16)
        IF A >= 3 THEN p_1 is an intersection pixel
    ENDIF
ENDFOR
```

Figure 8. Algorithm that identifies critical points in the skeleton when the intersection is really 4 pixels instead of only one.

by taking a skeleton pixel, p_s, and setting all of its eight neighboring pixels to the same label as p_s if the pixel on the skeleton raster is currently a background pixel and if that same pixel in the original binary image raster is a non-background pixel. Pixels are colored in a greedy manner so borders between different segments are colored based on which connected component region got there first. When complete, the union of all the segments will be exactly the original shape as shown in Figure 9.

The partitioning code is invoked in GeoGraphy through its chunking process, which was changed to *Geometric Operations* as a result of the integration of the partitioning code. In the next section, we will cover the process to invoke this process as part of a *Node Specification* in a *GeoQuestion* via the *GeoSearch* application.

StoredGraph Objects

The result of the path partitioning search is a collection of nodes stored to `NodeMatch_Table` within a StoredGraph containing one *Top-Level* node for each "match" and potentially several road segments as mid-level match nodes. Recall that this is a connected-components search, so each top-level node will be the same shape as the original node it came from, and the individual road segments will have pointers both from the original shape that they were derived from (the road node in `NodeRegionDurable_Table`) and to the top-level union shape. The nodes we will use in

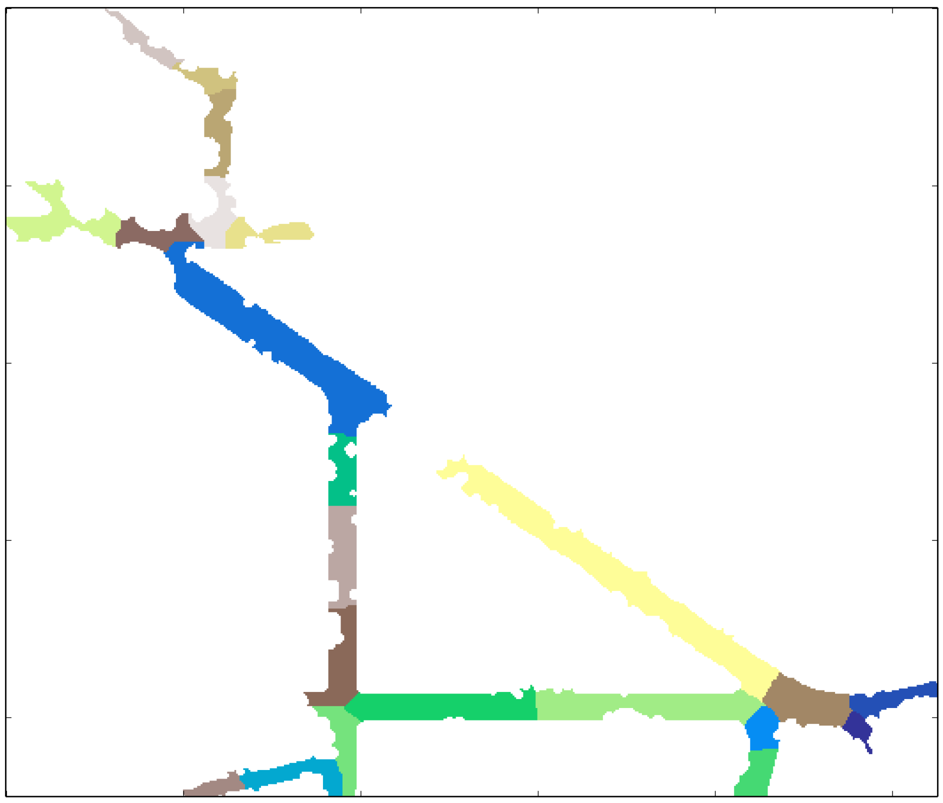


Figure 9. Shape after the *Shape Fill* step. Segments are shown using different colors.

future path connected searches will be the mid-level match nodes, since they are the individual road segments. See Figure B.2 in Appendix B for details. Each road segment is connected to adjacent road segments from the same original shape via *path-adjacent* edges. These are stored in `EdgePathAdjacent_Table` in the StoredGraph database.

Figure 10 illustrates how a simple road node might get partitioned. In this diagram, the lower image shows the original scene, and the upper image shows the StoredGraph objects generated by the path partitioning search and the relationships that are possible between them and other objects in the StoredGraph. The three objects in the original scene are *Road A*, *Sidewalk A*, and *Building A*. *Sidewalk A* directly touches *Road A* so there is an adjacency edge between them. *Building A* however, does not touch *Road A* so there are only distance edges between them.

After partitioning, we can see that *Road A* was partitioned into three pieces with separation at the intersection in the middle. There are *path adjacent* edges connecting *Road Segment A-1*, *Road Segment A-2*, and *Road Segment A-3* to one another. There are also *superset of* edges from the original shape to each segment to maintain the connection between them and their parent shape.

We also note that the relationship between *Sidewalk A* and *Road Segment A-1* and *Road Segment A-3* is now different. Even though they actually do touch, there are no *adjacency* edges between the sidewalk and the road segments. The primary reason for this is that recomputing the adjacency information between segments and external shapes would add extra overhead to our partitioning computation. This would be unnecessary, because a distance edge with a length of 0.0 can serve to associate both adjacent and non-adjacent nodes. If we identify a challenge problem in the future that specifically requires adjacency edges, then we can add the capability.

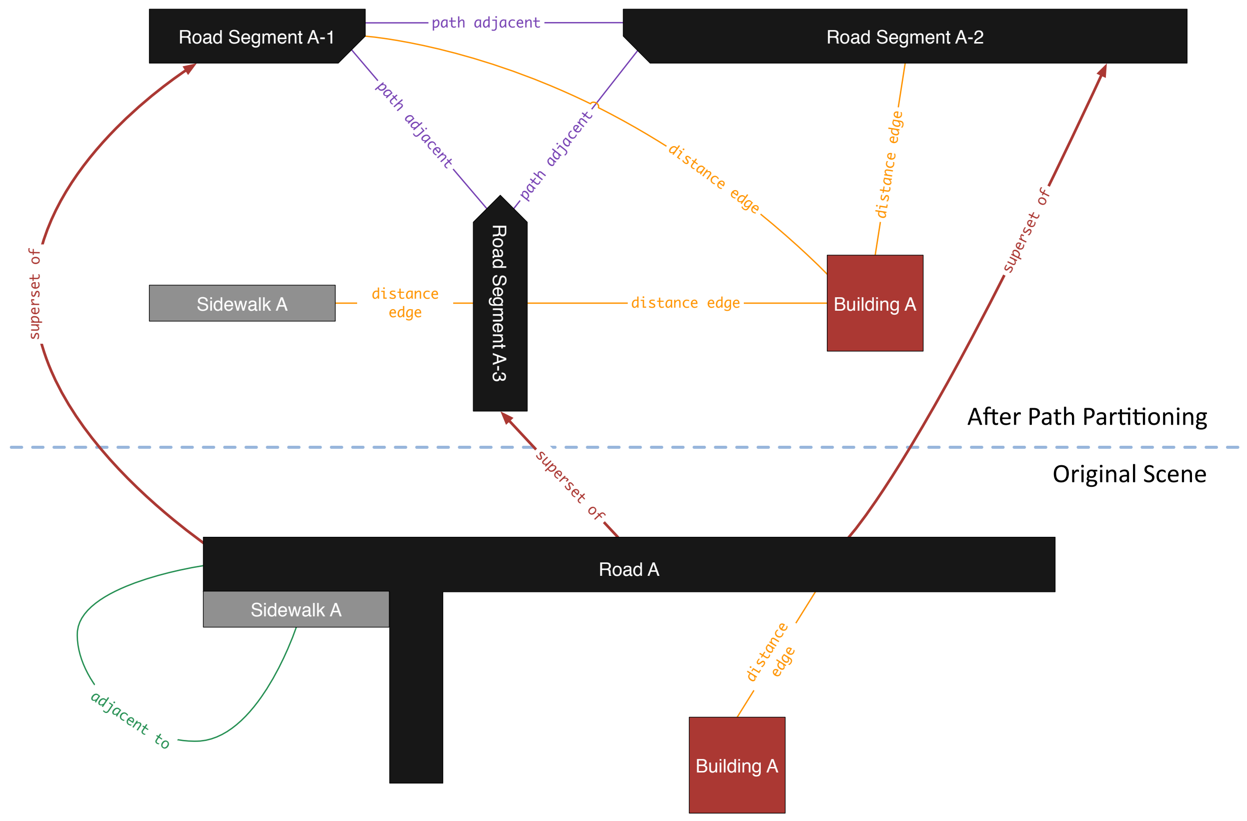


Figure 10. The StoredGraph objects resulting from a path partitioning search operation. It is worth noting that there are no *adjacency edges* between path segments and other nodes—path segments are only linked to other nodes in the graph via *distance edges*. Further, *path adjacent* edges are a new kind of edge generated in this search that link path segments to other path segments coming from the same partitioned shape.

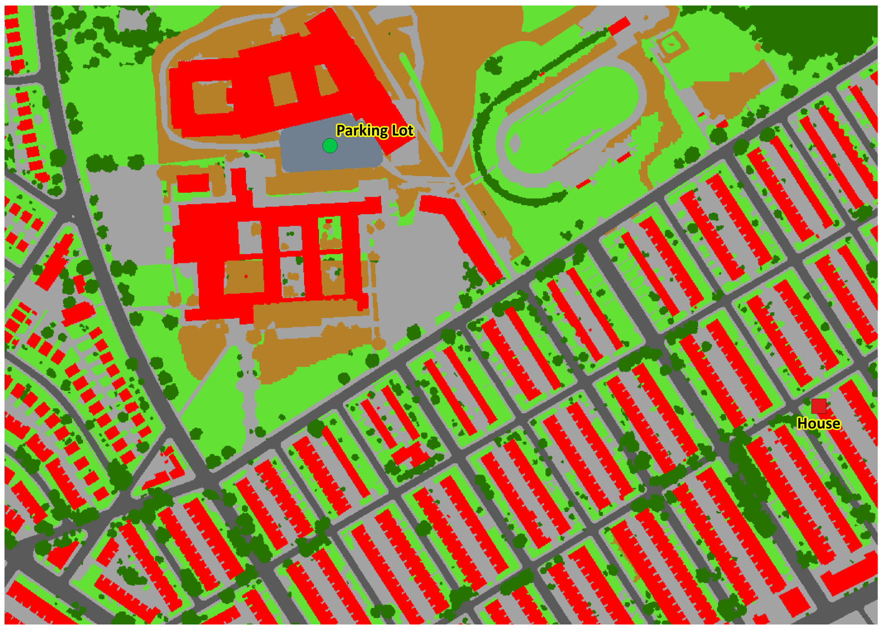


Figure 11. The land cover image for our example which will show analysis and search for a path from a parking lot to a house. The parking lot where the path starts is labeled in the middle. Buildings show up as red regions, roads are grey, grass and trees are light and dark green respectively.

Path Partitioning Usage

In this section we will describe how to use the GeoSearch program to execute the path partitioning operations described above. For our usage example, we will work through a scenario based on our Philadelphia data that we call the “Driving Home” example. In this example we are searching for a path that follows roads from a parking lot to a house. Figure 11 shows the scene for this problem. We provide an overview of the this example in Appendix A, on page 48.

GeoQuestion Basics

While a full description of the semantics of a GeoQuestion specification are out of the scope of this document, we provide a brief review of the components of a GeoQuestion. The general structure of a GeoGraphy experiment folder is organized in this manner:

Experiment Directory Structure

```
<prefix dir>/data
            /graph
            /search
                   /search_sequence_dir
                                       /1000_geoquestion_a
                                                          /input
                                                          /output
                                       /1001_geoquestion_b
                                       /1002_geoquestion_c
                                       ...
```

In GeoGraphy we use the GeoSearch application to execute queries against a StoredGraph. The GeoSearch application takes as its input a GeoQuestion, which consists of seven files:

SearchDefinition	Defines the search operation to perform (Star, Interrupted-Star, etc.).
Role	Defines the role to assign to search results.
NodeSpecList	Defines StoredGraph nodes to load into the SearchGraph.
EdgeSpecList	Defines edges for the SearchGraph.
RoleNode	Defines the roles of SearchGraph nodes for this problem.
RoleEdge	Defines the roles of SearchGraph edges for this problem.
PostProcessDefinition	Post-processing operations are specified in this file.

The specification files are simple ASCII text files in which lines can be comments, block begin/end, or key-value pairs. A hashtag (#) character is used to indicate the start of a comment—everything to the right of the # is ignored by the parser. Blank lines are ignored as well by the parser. Begin/End blocks are indicated by some tag prefixed by either "@begin_" or "@end_" and are used for special processing instructions for construction of the SearchGraph and for analysis. For additional details on how GeoQuestions are constructed, please consult the *Kitchen_Sink* example in the *Sample_Data* repository within GeoGraphy.

Matches from a GeoQuestion search are saved into `NodeMatch_Table` when write-back is enabled. Match nodes are stored in a hierarchical structure with top-level and mid-level match nodes with pointers from the original StoredGraph node to mid-level match nodes, which in turn have pointers to the top-level match node for the match. Further information on match hierarchy can be found in Appendix B.

The following seven sub-sections will provide details on each of the GeoSearch input files as they are currently used for a path partitioning operation.

Search Definition

The Search Definition file provides information about the type of search that is being performed. Search methods can be "Connected_Components," "Star_Graph," etc. For our path partitioning problem, the search definition file looks like:

SearchDefinition.csv

```
# SEARCH DEFINITION

Search_Method, Connected_Components
Minimum_Component_Size,1
Maximum_Component_Size,1000000000
```

In this case, the **S**earch Method is a *Connected Components* search, meaning that each "match" is simply the connected component of the search graph without any further pattern-matching constraints applied. The Minimum_Component_Size constraint is set to 1 so that we keep matches that have at least one node. The Maximum_Component_Size constraint is set to 1,000,000,000 to effectively set the upper limit to infinity for this search.

Role Specification

The role specification file assigns the role to the *top level match node* from this search. Match roles are stored in `RoleNode_Table` within a StoredGraph. This field designates the role of a match from this search when it is stored back to the StoredGraph. Match nodes are saved into `NodeMatch_Table` and we join `NodeMatch_Table` with `RoleNode_Table` by linking `NodeMatch_Table.Role_ID` to `RoleNode_Table.RoleNode_ID`.

The role specification file is a CSV file which specifies one role entry per line. Each role entry has five parameters: the name, short name, color, fill pattern, and explanation. The *name* field is the name of the role in long form. The *short name* field is just a shortened name of the role for convenience and labeling. the *color* field is a named color that must match the list of colors in the StoredGraph, which come from the `GeoGraphy_Color.csv` file in the data directory, which is input during `AddToStoredGraph` execution. *Fill pattern* is currently unused but is a placeholder for adding fill patterns in addition to colors for rendering purposes. Currently fill pattern is always set to "No texture." Lastly, the *explanation* field is a long description of the role that is primarily used for rendering but is also very useful for GeoQuestion developers in providing documentation about the purpose of the role. In a `Role.csv` file there will be only one role entry. The format is the same for the `RoleNode.csv` file, which contains multiple roles and is described in Section 2.3.

In this GeoQuestion the result of our search is a partitioned road network, so we set the Name field to "Road Network" and will color it Gray.

Role.csv

```
#Name             Short                Fill
#(No Commas)      Name       Color     Pattern         Explanation
#-----------------------------------------------------------------
"Road Network",  "RDNet",   "Gray",   "No texture",   "Road Network"
```

Role Node Specification

The `RoleNode.csv` specification file is formatted in the same manner as the role specification file, but the roles described are used in the search operation and are assigned to nodes in the node specification file, which will be described in Section 2.3. These roles are also saved in the StoredGraph if write-back is enabled.

The role node specification for our example is:

RoleNode.csv

```
#Name                     Short                            Fill
#No Commas)               Name                 Color       Pattern        Explanation
#---------------------------------------------------------------------------------------------
"Road Segment (20;80)",   "RDseg(20;80)",      "Lavender", "No texture",  "Road segment ..."
```

We only have one node type in this search, so there is only need for one node role to be specified. Since this role is the one that will be assigned to the *individual road segments* after partitioning, we use a suggestive labeling scheme to encode our parameterization to the partitioning algorithm. In this search, we will set the minimum segment length to 20 skeleton pixels and the maximum segment length to 80 skeleton pixels, meaning that after intersections are removed we will erase any skeleton segments shorter than 20 pixels. After that, any skeleton segments longer than 80 pixels will be evenly partitioned so that none are longer than 80 pixels in length. These parameter values are reflected in the (20;80) suffix on the name field, for documentation purposes. (The actual control parameter values will be specified in `NodeSpecList.csv`, described in Section 2.3.) The naming scheme is arbitrary, but we have used this scheme in numerous GeoGraphy searches as an internal standard way to label nodes resulting from geometric operations.

NodeSpecList Specification

The `NodeSpecList.csv` file is where most of the action is happening in this GeoQuestion. The NodeSpecList file provides the SQL search constraints to extract nodes from the StoredGraph and assigns the proper SearchGraph roles to these nodes. We use the following NodeSpecList for our scenario:

NodeSpecList.csv

```
1  # NODE SPEC LIST DEFINITION
2  #
3  @begin_Node_Spec
4
5  Role_Name, Road Segment (20;80)
6  Table_Name, NodeRegionDurable_Table
7
8  @begin_SQL_Query
9  SELECT NodeRegionDurable_Table.Node_ID
10 FROM NodeRegionDurable_Table
11 WHERE content=(SELECT RegionContent_ID
12                FROM RegionContent_Table
13                WHERE Name = 'Road')
14        AND CS_Area > 100
15 ORDER BY CS_Area DESC
16 @end_SQL_Query
17
18 # Paved areas are often connected by thin paths. Break these up.
19 @begin_Geometric_Operation_Spec
20 Operation_Type, Path
21 Min_CC_Size,    20         # Min CC Size for the algorithm
22 Max_CC_Length,  80
23 @end_Geometric_Operation_Spec
24
25 @end_Node_Spec
```

In our search, we have only one node spec, which fetches the road regions. In this case, `Role_Name` will be the role specified for the individual segments (mid-level match nodes) and not the top-level match nodes (See Appendix B). The value for the `Role_Name` field is specified in the role node specification file described above.

On line 5 we declare the role name to assign to the partitioned segment nodes; this corresponds to the role specified in the RoleNode.csv file. Line 6 contains the table name of the source table in the StoredGraph that these nodes will come from. In our case, we are partitioning road-network nodes which are durable region nodes, so we look for them in `NodeRegionDurable_Table`. Lines 8-16 define the SQL query used to extract road nodes from the StoredGraph that we want to partition. In our case, we are simply pulling all nodes from `NodeRegionDurable_Table` that are "Road" nodes.

Partitioning is a geometric operation on the shape, because we are generating new nodes through some morphological operation on existing shapes. The partitioning process is invoked in the Geometric_Operation_Spec block, lines 19-24, and the actual operation is applied during SearchGraph node construction.

The Geometric_Operation_Spec block can perform many different operations, and is the catch-all for morphological operations on shapes loaded into the SearchGraph. The first key/value pair expected is `Operation_Type`, which specifies what kind of geometric operation is being performed. We want path partitioning, so we use "Path" as the value on line 20.

Next we provide the two parameters for the partitioning process on lines 21 and 22. Recall that we want to keep skeleton segments that are at least 20 pixels in length and break up long segments to be 80 pixels long or shorter. The `Min_CC_Size` parameter specifies the minimum length constraint on the skeleton segments and the `Max_CC_Length` parameter provides the maximum segment length for the algorithm.

There are additional parameters that can be given in the geometric operation specification, such as `Min_Area`, that are not part of the path partitioning process itself. These may be applied as post-operation constraints on the shapes generated by the operation to further constrain what gets into the SearchGraph. An in-depth description of these additional constraints is outside the scope of this document.

Role Edge Specification

The `RoleEdge.csv` specification file is another labeling file in the same manner as the `Role.csv` and the `RoleNode.csv` files. As the name would imply, this one is for SearchGraph edges. In most cases we do not need to modify this file since most SearchGraph edges are distance edges which are a standard edge type. The standard `RoleEdge.csv` file is the following:

RoleEdge.csv

```
#Name              Short
#No Commas)        Name      Color       Explanation
#---------------------------------------------------------------------------
"Adjacent-To",     "Adj",   "White",     "Features share a common border."
"Proximity",       "Px",    "Cyan",      "Features within distance limit."
"Changed-To",      "CT",    "Yellow",    "One feature replaced another."
```

Note that the path partitioning operation does produce *path-adjacent* edges as part of the search. These are not enumerated in this file, because they are hardcoded into the GeoGraphy library itself as a special edge type since they are generated as part of the partitioning operation.

Edge Spec List Specification

The `EdgeSpecList.csv` file normally provides constraints on the edges involved for a search. In the path partitioning specification we don't need to specify any edges, so this file is left blank.

EdgeSpecList.csv

```
# EDGE SPEC LIST DEFINITION
```

Postprocess Definition

The last file in a GeoQuestion specification is `PostprocessDefinition.csv`. As the name would imply, this file contains specifications for post-processing operations that are applied to match candidates after the search is performed, as a final acceptance or rejection step on matches. The `PostprocessDefinition.csv` file for our example follows:

PostprocessDefinition.csv

```
# POSTPROCESS DEFINITION

@begin_Postprocess_Match_Spec_List

@begin_Postprocess_Match_Spec
Postprocess_Match_Method, None
@end_Postprocess_Match_Spec

@end_Postprocess_Match_Spec_List
```

This file is essentially a no-op because we have no need for post-processing constraints in our road partitioning search.

Invoking Partitioning Search

Execution of a search is performed using shell scripts which are invoked from the root search directory for a specific search. Detailed search invocation instructions are outside the scope of this document, but we will provide a brief overview of the basic procedure for standard search environments.

The standard path layout will have the GeoQuestion specification files in a `<search_name>/input` directory. If your search is generated by first copying in an existing standard search and customizing it to your applications, there should be a `run_<search_name>.sh` file in the `<search_name>` directory. Provided you have your environment set up properly with the correct paths and environment variables pointing to your GeoGraphy installation, executing the script will run the search. Results of the search will be put into the `output` directory. Typically, we use QGIS [19] to visualize our results.

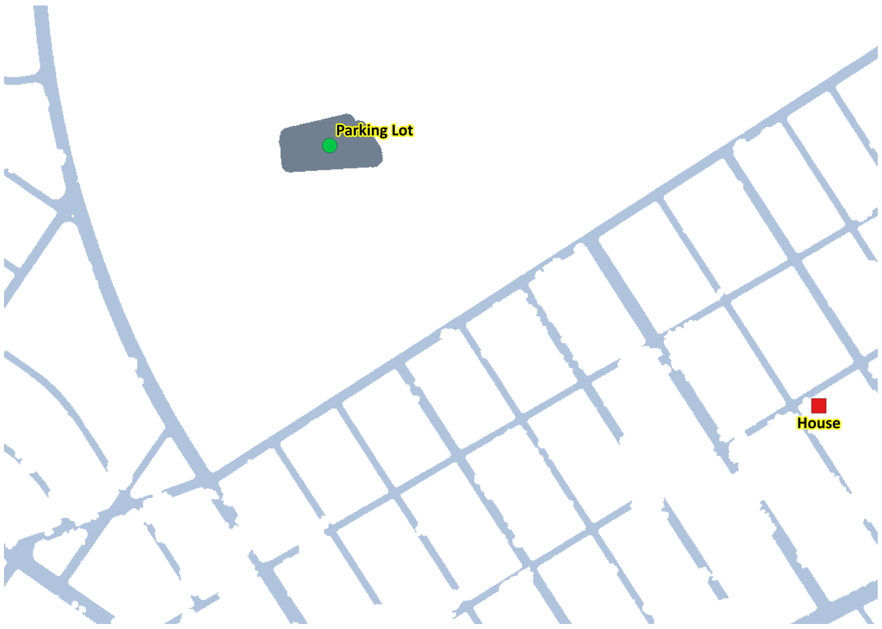


Figure 12. The road network from Figure 11. Contiguous regions of pixels are individual road segments. For our example all road nodes are partitioned.

Result of Path Partitioning Search

Figures 11,12,13, and 14 show images from QGIS of the “Driving Home” problem. Figure 11 illustrates the scene for our problem by showing the LandCover image set with the *parking lot* and the *house* labeled. Figure 12 shows just the road network from the scene.

After partitioning is completed, the graph of road segments and path-adjacent edges connecting them is shown in Figure 13 with just the graph and start/destinations labeled, and in Figure 14 with the landcover background added for context. Looking closely at the path network graph, you will notice that not all four-way intersections will be fully connected. This occurs as an artifact of the partitioning process and the pixel-based rasters when opposite paths interface and “touch” in the middle, separating opposite paths from directly touching.

One solution to this problem might be to modify the partitioning code to perform a more sophisticated operation at intersections to segment the shape rather than simply breaking the skeleton at the intersections. EDT information could be used to give insight into the thickness of the shape at each position in the skeleton, which could be used to clean up these artifacts if necessary.

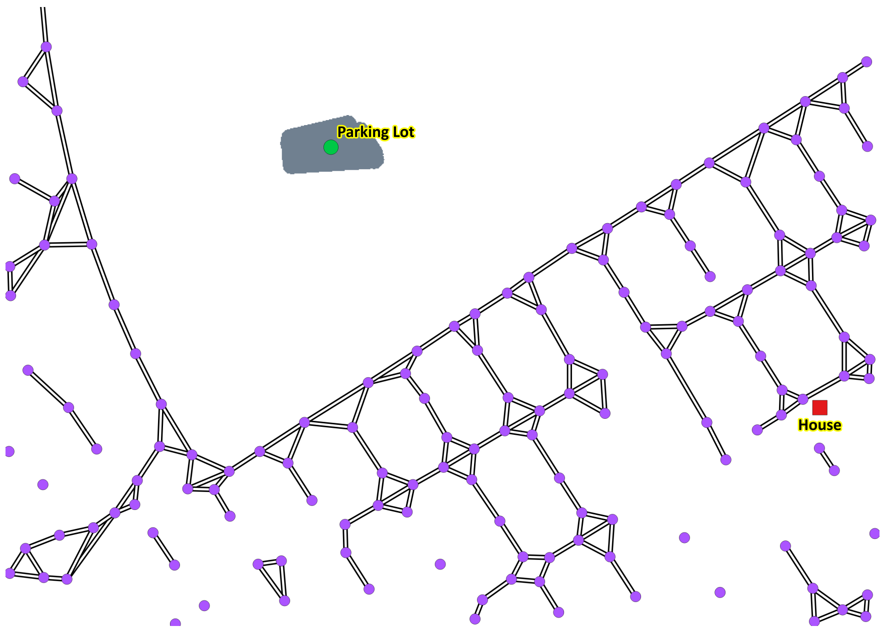


Figure 13. After partitioning is completed the segment nodes and path-adjacency edges are written to the StoredGraph. This figure shows the path segment nodes and their path-adjacency edges. Note: the point locations for the nodes are set to the *representative point* of each segment, which will be an interior point near the middle of each segment.

Path Connected Search

Path Connected Search Background

The GeoGraphy library has traditionally used *distance* edges almost exclusively in its searches. Distance edges encode the straight-line distance between two nodes in the GSTSG and are good at capturing the proximity between two objects, but they do not store any context of what might lie between the nodes. They also cannot capture information about paths between locations that might be interesting to an analyst. Additionally, when investigating activity data one would like to be able to perform reachability type queries of the form “Is there a valid path between Building A and Building B that could have been traversed?” or “Where can I get if I start at Building A and follow only sidewalks and roads?” We cannot solve problems like these using distance edges alone, which motivated the need for the *path-connected edge*.

We chose to implement this with a simple ST (Source to Target) search that will find a single shortest path between two nodes, following only node segments that came from a path-partitioning step. The Dijkstra weighted single source shortest path algorithm [7] provides an optimal solution and executes in linear time. We currently weight hops along a path evenly but can change the weighting in the future to include temporal or distance-traveled constraints.

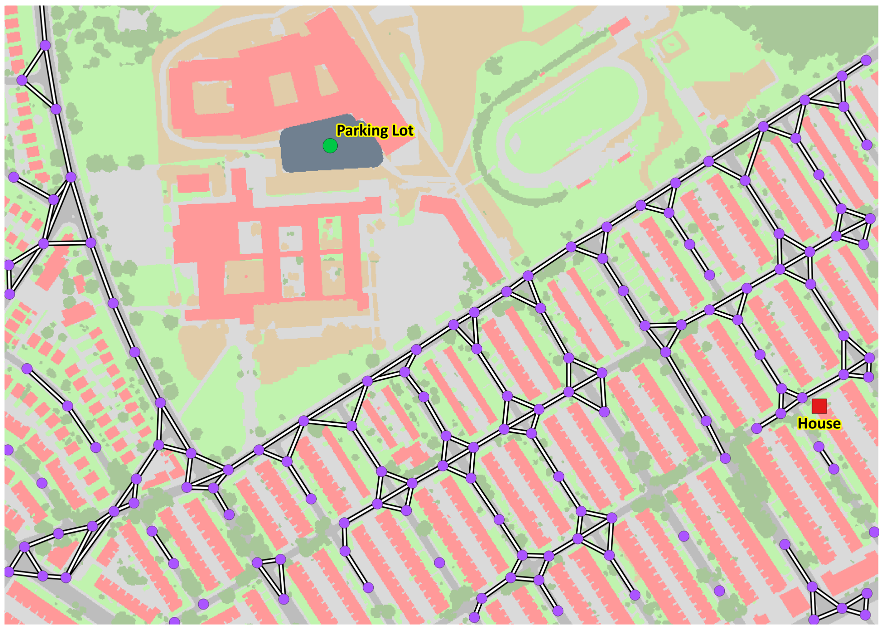


Figure 14. The original scene with the parking lot and road network overlaid, after path partitioning has been completed.

In the future, we would like to expand our path-connected search capabilities to include not just the shortest path but to find multiple alternative shortest path candidates. We can also add further constraints on road/path width, distance, and possibly time constraints. In the future, we might explore additional algorithms that go beyond a simple Dijkstra shortest path search such as an A* shortest path search [5] or "connection subgraphs" algorithms [10].

Path Connected Search Technical Details

Integrating the *path connected* search into GeoGraphy resulted in the addition of a new paradigm of edge construction for edge specifications in a GeoQuestion. The need to implement a path-aware search in GeoGraphy that would integrate well into the existing star-graph and interrupted star search models imposed constraints on the design of what the path-connected search would generate and thus how it might operate. The star graph search model has historically used a proximity model using distance edges between two nodes, A and B. For our search to integrate into existing GeoGraphy search capabilities we must define a new edge type which will directly link A and B, but have a different semantic meaning from the traditional distance edge.

We solved this by creating the *path-connected edge* type. One can read a path-connected edge as "Two nodes, A and B, are said to be path-connected if a valid path exists from A to B following the constraints provided in the GeoQuestion that generated the edge." Figure 15 illustrates the overall structure of what a path-connected edge represents.

The addition of the path-connected edge introduced a new kind of edge class into the GeoGraphy ecosystem: the *derived edge*. A derived edge is an edge that is derived from some topological information about the graph during stored graph edge construction.

Derived Edges

The path-connected search problem generated a new edge generation abstraction for the GeoGraphy code. We observed that a path-connected edge follows a completely new paradigm of generation from any previous edges in the GeoGraphy code. These edges are different because they are generated through topological analysis of the graph at search time.

The GeoGraphy code is a research code, and thus has always been evolutionary; whenever possible we have tried to build in flexibility in anticipation of when (not if) new capabilities will be necessary to develop. This mindset led us to think of a slightly bigger picture than simply path-connected edges. We see these as representing a whole new class of edge type that we call derived edges, because they are edges that result from some analysis of the graph structure. Under this model, path-connected edges are a specific type of derived edge.

As mentioned earlier, the use of Dijkstra's shortest path search to generate our path-connected edges is one of many possible algorithms we can use to generate a path between two nodes in a graph. To accommodate future algorithm options for path-connected edges we add another level to our hierarchy that allows us to specify the algorithm to use for path-connected edge generation. We will provide additional information when we cover the `EdgeSpecList.csv` specification in Section 3.3.

Path Connected Edges

Path-connected edges are stored in `PathConnectedEdge_Table` and have the usual edge properties plus an `Encoded_Path` column that stores a delimited string encoding the primary keys of the nodes and edges that make up the path. The encoded path field can be used to recover the actual path taken between the nodes and is currently used in the rendering code to draw the exact path taken. The format of an encoded_path string is the following:

Encoded_Path field in PathConnectedEdge_Table

```
6,20,116;18,78;6,20,121;18,86;6,20,127;  ... ;6,20,175;18,160;6,20,181
```

The string is delimited in two levels that should be understood for parsing. The first level is delimited by semicolons ";" which separates the string into a list of alternating node and edge ids: `<node_id>;<edge_id>;<node_id>;...;<node_id>,<edge_id>;<node_id>`. The first and last entry will always be node ids and there is always an edge between nodes. Nodes and edges can further be tokenized by splitting on commas into the *role_id*, *table_id*, and *node_id*. These three values are necessary in GeoGraphy to accurately look up match nodes, which is the kind of node that path segments will be by definition. Edges have only two values, the *table_id* and *edge_id*.

Figure 15 illustrates what is represented by a path-connected edge. In this example we have a path network node that has been partitioned into path segments in a prior partitioning operation and

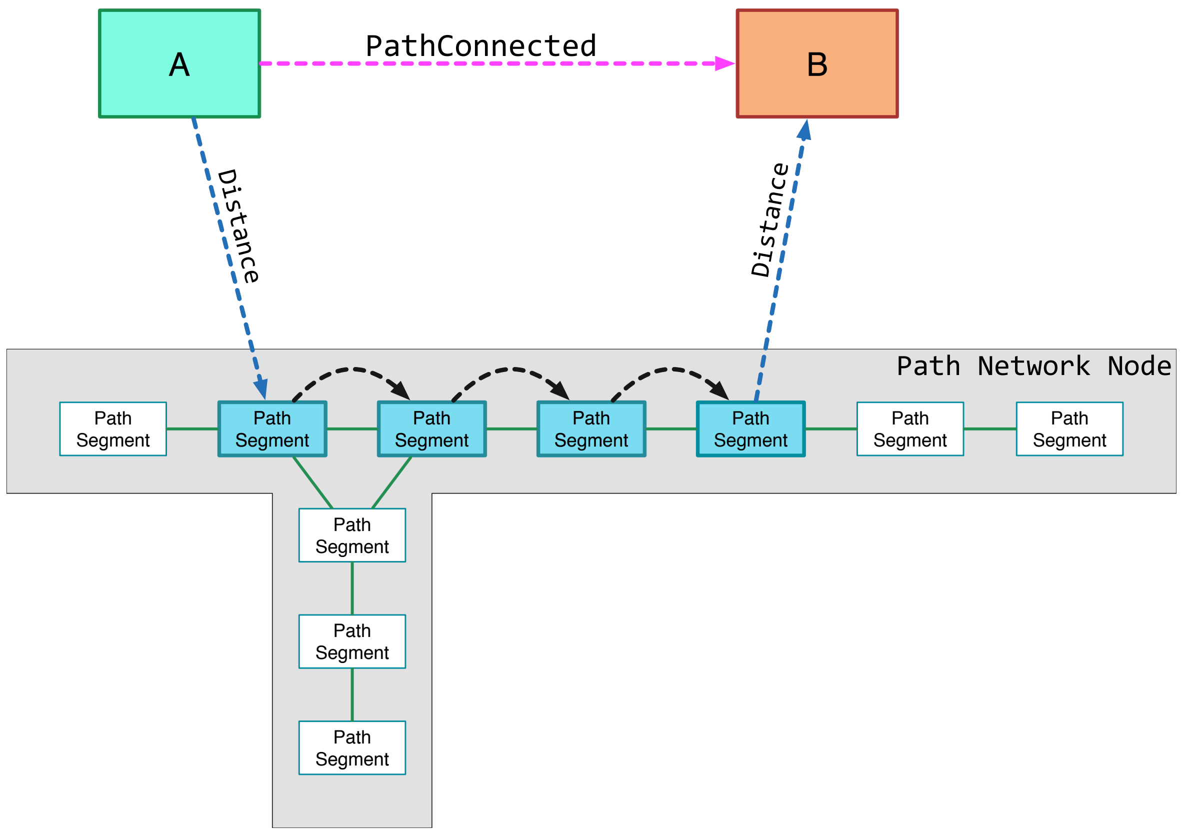


Figure 15. A path-connected edge provides a direct link between *A* and *B* and can be interpreted as "A path between *A* and *B* using a path network which has been through the path-partitioning operation exists that meets the constraints provided in the GeoQuestion that generated it."

two other nodes, *A* and *B*. This diagram illustrates a new wrinkle to the path-connected search problem: In addition to finding the shortest path via a partitioned path network, we must also hop *onto* the path from *A* and hop *off of* the path to *B*. These path-entry and path-exit edges must be distance edges, since our model is to use distance edges to link path and non-path nodes. Further, we might want to specify constraints on distance or direction of the path-entry or path-exit edges, which would add complexity to the problem specification in our GeoQuestion. Lastly, we want to constrain the actual path followed through the path segments to a maximum number of hops.

All of this complexity gets wrapped into a single edge that we call the path-connected edge. It is a derived edge and tells the GeoSearch algorithm that "there is a valid path between *A* and *B*, that satisfies the constraints specified in the GeoQuestion that generated it." By compressing this information into a single edge, Star and Disconnected-Star searches in GeoGraphy can now use path-connected edges in their searches without modification.

Path Connected Search Usage

The search specification for a path-connected edge search is more complicated than the path-partitioning search since there are multiple node types involved. In this search example we are using a Star search and will set the hub as the *START* node in the path and the spoke node will be the *FINISH* node.

We will also need to load the path-segment nodes into the SearchGraph, but these will not participate in the search directly. They are needed because construction of the path-connected edge happens during SearchGraph edge construction. SearchGraph edges are constructed *after* SearchGraph nodes, so when GeoSearch gets to the edge construction phase it will have all the necessary nodes in the SearchGraph for construction of the path-connected edge. After SearchGraph construction is complete the path-connected edges will be accessible to the core star (and disconnected-star) search algorithms.

The following sections will cover each of the seven input files for our path-connected GeoQuestion in the same manner as the path-partitioning search was described earlier in this document. For the sake of brevity, we will not provide in-depth explanations of the formatting of files that were previously addressed in the path-partitioning search specification.

Path Connected Search Definition

The SearchDefinition file sets this search up as a Star Graph search with the HUB node set to `Path Start` with one spoke node, `Path Target`. These role names are specified in the RoleNode specification file (see Section 3.3). The `Minimum_Required` and `Maximum_Required` fields in lines 23 and 24 are arbitrary but the minimum should be at least one.

SearchDefinition.csv

```
1  # ========================================================================
2  # SEARCH DEFINITION
3  # ========================================================================
4  
5  # ------------------------------------------
6  # Search Method
7  # ------------------------------------------
8  Search_Method, Star_Graph
9  Hub_Role_Name, Path Start
10 Hub_Interrupt_Role_Name, NO_INTERRUPT
11 
12 # ------------------------------------------
13 # Spoke definitions (Optional)
14 # ------------------------------------------
15 @begin_Spoke_Spec_List
16 
17 # ----------
18 # Spoke
19 # ----------
20 @begin_Spoke_Spec
21 Spoke_Role_Name, Path Target
22 Spoke_Interrupt_Role_Name, NO_INTERRUPT
23 Spoke_Minimum_Required,  1
24 Spoke_Maximum_Allowed,   10
25 @end_Spoke_Spec
26 
27 @end_Spoke_Spec_List
```

Path Connected Role Specification

Role.csv

```
#Name                     Short             Fill
#No Commas)               Name     Color    Pattern       Explanation
#-------------------------------------------------------------------------------------
"Path Connected Search",  "PCS",  "Magenta",  "No texture",  "Path-Connected Search"
```

The role specification sets the role of the resulting top-level match node. In this case we simply label it a “Path Connected Search” result.

Path Connected RoleNode Specification

There are three node roles for SearchGraph objects on this search. The `Path Start` and `Path Target` role names are the start and finish nodes in the graph for our path search. The `Road Segment TEMP` node role will be the road segments from the partitioning search we described earlier in this document. Specifically, these nodes will be the match nodes with the role `Road Segment (20;80)` assigned to them.

RoleNode.csv

```
#Name                       Short                            Fill
#No Commas)                 Name             Color          Pattern       Explanation
#---------------------------------------------------------------------------------------------
"Path Start",               "S",             "DarkGreen",   "No texture", "Start node"
"Path Target",              "T",             "DarkRed",     "No texture", "Target node"
"Road Segment TEMP",        "RoadSegTEMP",   "Black",       "No texture", "Road Segment"
```

Path Connected NodeSpecList

The NodeSpecList file contains the specifications to find our three node types in the StoredGraph and add them to the SearchGraph. Since there are no geometric operations performed on these nodes, the individual *Node_Spec* blocks are essentially just queries into the StoredGraph.

NodeSpecList.csv

```
# NODE SPEC LIST DEFINITION
#
# Paved Chunk -- AnneArundel

# ----------------------------------------
# Starting point : Parking Lot
# ----------------------------------------
@begin_Node_Spec

Role_Name, Path Start
Table_Name, NodeMatch_Table

@begin_SQL_Query
SELECT NodeMatch_Table.Node_ID
FROM NodeMatch_Table
WHERE Role_ID=(SELECT RoleNode_ID
               FROM RoleNode_Table
               WHERE Name = 'Parking Lot')
@end_SQL_Query

@end_Node_Spec


# ----------------------------------------
# Starting point : House
# ----------------------------------------
```

```
27 @begin_Node_Spec
28
29 Role_Name, Path Target
30 Table_Name, NodePointDurable_Table
31
32 @begin_SQL_Query
33 SELECT NodePointDurable_Table.Node_ID
34 FROM NodePointDurable_Table
35 WHERE "Feature Name" = "House"
36 @end_SQL_Query
37
38 @end_Node_Spec
39
40
41 # ---------------------------------------
42 # Path Network Nodes
43 # ---------------------------------------
44 @begin_Node_Spec
45 Role_Name, Road Segment TEMP
46 Table_Name, NodeMatch_Table
47
48 @begin_SQL_Query
49 SELECT Node_ID
50 FROM NodeMatch_Table
51 WHERE Role_ID IN (SELECT RoleNode_ID
52                   FROM RoleNode_Table
53                   WHERE Name IN ("Road Segment (20;80)"))
54 @end_SQL_Query
55
56 @end_Node_Spec
```

The `Path Start` node spec is contained in lines 8-21. This is the parking lot that our path will begin from. In this case, the source table for this node is `NodeMatch_Table`, because the full search sequence performed a chunking operation to separate the parking lot from other parking lots where they are connected by thin sidewalks, etc. Full details of the chunking operation are outside of the scope of this document, but a more detailed overview of the sequence of searches that make up the "Driving Home" example is provided in Appendix A.

Next, the `Path Target` node spec can be found on lines 27-26. The notable detail on this node is that it is not a *region* node at all, but rather it is a *durable point* node. This node was loaded into the StoredGraph via a Comma Separated Variable (CSV) file input to the AddToStoredGraph application. We use this node type in our sample search to further demonstrate the flexibility of the GSTSG by mixing heterogeneous node types.

Lastly, the `Road Segment TEMP` node specification is located on lines 44-54. This node specification contains the *path segment* nodes that we will allow our path-connected edge to follow. Recall, these nodes must be the results from a prior path-partitioning search and will be found in `NodeMatch_Table` within the StoredGraph. In our case, they correspond to the `Road Segment (20;80)` role, and were written to the StoredGraph as mid-level match nodes by the path partitioning search described in Section 2. These nodes are unusual, because they do not participate directly in the search and are not even part of the star-graph search specification that was shown in the `SearchDefinition.csv` file. We include them in the node specification because we can leverage the fact that SearchGraph nodes are loaded before SearchGraph edges are generated. This has the following advantages:

1. We leverage the constraint logic already in place to specify nodes and load them into the SearchGraph.
2. It simplifies the burden on the `EdgeSpec.csv` file to find the node type through which we can follow a path.
3. It simplifies the lookup requirements during the path-connected edge generation to find these nodes since they will already be loaded into the SearchGraph.

Of these reasons, (1) is probably the most important one because a tremendous amount of development has gone into SearchGraph node construction, including the SQL interface design and constraint logic.

For example, should we need to implement a search in the future that might look for road paths that exceed a minimum width that could accommodate a large truck (i.e., avoid alleyways, etc.), we could provide that constraint in the NodeSpec to give us only those path-segment nodes that met our minimum width requirement. This would allow only path-segment nodes that exceed the minimum width in the SearchGraph; nodes that aren't wide enough would be dropped and thus insert "breaks" in the network. As a result, the Dijkstra shortest path code would only be able to search for paths on roads wide enough for our truck to pass through. [1]

Path Connected RoleEdge Specification

The `RoleEdge` specification file is purely boiler plate at this time and uses the standard GeoGraphy RoleEdge template.

RoleEdge.csv

```
#Name                  Short
#No Commas)            Name      Color           Explanation
#------------------------------------------------------------------------------------------
"Adjacent-To",         "Adj",  "White",        "Features share a common border."
"Proximity",           "Px",   "Cyan",         "Features within distance limit."
"Changed-To",          "CT",   "Yellow",       "One feature replaced another."
```

Path Connected EdgeSpecList

In the path-connected search the `EdgeSpecList.csv` file file is the most crucial file. This is where we instruct GeoSearch to construct a path-connected edge during the edge construction phase of

[1] In reality, we could not support this search at this time because width metrics are not saved to path segment nodes, but we could add this easily with the addition of Euclidean Distance Transform (EDT) filter data onto the skeleton. We merely cite this example to illustrate how the powerful constraint logic that is already implemented in the NodeSpec handler could be leveraged in a path-connected search.

SearchGraph construction. Similar to the Geometric_Operation in a NodeSpecList file from the path-partitioning problem, we have the basic constraints and parameters for our edge as well as an `@begin_` and `@end_` block that specifies the specialty operation.

The `EdgeSpecList.csv` listing for our example case is the following:

EdgeSpecList.csv

```
# EDGE SPEC LIST DEFINITION

# ------------------------------------
# Edge Spec
# ------------------------------------
@begin_Edge_Spec

Edge_Role_Name, PathConnected-To
Edge_Table_Name, EdgePathConnected_Table
Node_Role_Name_1, Path Start
Node_Role_Name_2, Path Target

# Symmetric Constraints
Maximum_Allowable_Minimum_Distance, 6000
Minimum_Allowable_Minimum_Distance, 100

#----------------------------
# Derived Edge Specifications
#----------------------------
@begin_Derived_Edge_Spec

# Derived_Edge_Type
#   - What kind of derived edge will this be? Options are:
#      a) PathConnected
Derived_Edge_Type, Path_Connected

# Algorithm
# Options:
#   a) SingleST - returns a simple ST shortest path between S & T.
Path_Algorithm, SingleST

# Path_Node_Role_Name
#   - What Node Roles will be the nodes we traverse to find the path?
Path_Node_Role_Name, Road Segment TEMP

# Symmetric Constraints
Maximum_Allowable_Minimum_Distance_N1_to_Path, 200
Maximum_Allowable_Minimum_Distance_N2_to_Path, 50

Minimum_Allowable_Minimum_Path_Distance_in_Hops, 0
Maximum_Allowable_Minimum_Path_Distance_in_Hops, 100
@end_Derived_Edge_Spec

@end_Edge_Spec
```

The edge specification for our edge is inside the `@begin_Edge_Spec` and `@end_Edge_Spec` tags on lines 6 and 44. The first set of keys specify the general information about the path-connected edge that will be constructed and global constraints on the possible edges. These are shown in lines 8 through 15 in our example.

The `Edge_Role_Name` field on line 8 should be set to `PathConnected-To` as this type is hard-wired into the GeoGraphy code for this particular search. Next, the `Edge_Table_Name` key will need to be set to `EdgePathConnected_Table` to match the table of the same name in the StoredGraph. This is where the new path-connected edges will be written.

The next two lines, 10 and 11, are the node role names for our start and end nodes. These node roles will correspond to the nodes that were created in `NodeSpecList.csv`, shown earlier in Section 3.3 with the roles `Path Start` and `Path Target`. `Node_Role_Name_1` is the key for the starting node in our search, so we assign it to `Path Start`. Next, `Node_Role_Name_2` is the key for the finish node, so this one is assigned to `Path Target`.

The next set of keys on lines 14 and 15 specify some general symmetric constraints on our edge pairs. These set the minimum and maximum distance between any pair of candidates to consider. In our case the `Maximum_Allowable_Minimum_Distance` key provides the upper bound and the `Minimum_Allowable_Minimum_Distance` key is the lower bound on the shortest distance between `Path Start` and `Path Target`. Any candidate `Path Start` and `Path Target` nodes must be within these distance constraints before they can even be considered for a path-connected edge. This is advantageous because the path-connected search is computationally expensive compared to a distance comparison.

After this the `Derived_Edge_Spec` block is defined on lines 20-42. Inclusion of this block of instructions in our GeoQuestion specification instructs GeoGraphy to construct this edge as a path-connected edge. The keys for this block are:

Derived_Edge_Type (line 25) The kind of derived edge we will create. In this case, we are creating a `Path_Connected` edge. Currently there are no other kinds of derived edges that can be created in GeoGraphy, but we have designed the code to allow for future types.

Path_Algorithm (line 30) The algorithm to use to construct this edge. This is set to `SimpleST`, which invokes the Dijkstra shortest path search for our path finding algorithm. Edges are uniformly weighted so that we will find the shortest path in terms of number of hops from start to finish. Finally, the SimpleST search returns only one path even if there are more than one possible shortest paths.

Path_Node_Role_Name (line 34) What kind of path nodes can this path follow? This value corresponds to the role specified in the `NodeSpecList.csv` file for the path-segment nodes we selected from the previous path partitioning operation. In this case, we set it to `Road Segment TEMP`.

Maximum_Allowable_Minimum_Distance_N1_to_Path (line 37) Upper bound on distance from `Path Start` to the first path node in meters. Side effect: this creates distance edges between the `Path Start` node and candidate path-segment nodes.

Maximum_Allowable_Minimum_Distance_N2_to_Path (line 38) Upper bound on distance from the last path node to `Path Target` in meters. Side effect: this creates distance edges between the `Path Target` node and candidate path-segment nodes.

Minimum_Allowable_Minimum_Path_Distance_in_Hops (line 40) Lower bound on the path length in terms of path-adjacent edges traversed.

Maximum_Allowable_Minimum_Path_Distance_in_Hops (line 41) Upper bound on the path length in terms of path-adjacent edges traversed.

Path Connected Postprocess Definition

Finally, the `PostprocessDefinition` file is provided to finish our GeoQuestion specification. In this search there are nothing to do in post processing, so we use the standard GeoGraphy "no-op" `PostprocessDefinition` file template.

PostprocessDefinition.csv

```
# POSTPROCESS DEFINITION
@begin_Postprocess_Match_Spec_List

@begin_Postprocess_Match_Spec
Postprocess_Match_Method, None
@end_Postprocess_Match_Spec

@end_Postprocess_Match_Spec_List
```

Invoking Path Connected Search

Invoking the path-connected search is done in the same manner as the path-partitioning search we described earlier in Section 2.3.

Path Connected Search Results

After the search has run the resulting output will contain path-connected edges if there were any valid paths found between the specified start and finish nodes. In our example, there is one "shortest" path from the parking lot to the house.

Figures 16 and 17 show the resulting path found by the Dijkstra shortest path algorithm. In Figure 16 we include the land cover imagery in the background for reference. Recall from the path-partitioning example that the white edges correspond to the path-adjacent edges which connect two path segment nodes. The path found is shown by the black line with red dashes. The first and last edges in the sequence are the *entry* and *exit* edges that connect the start and finish nodes in the path.

The path-connected edge is shown in figures 18 and 17. Again, the first image shows the land cover image in the background with the graph overlaid and the second image shows only the graph

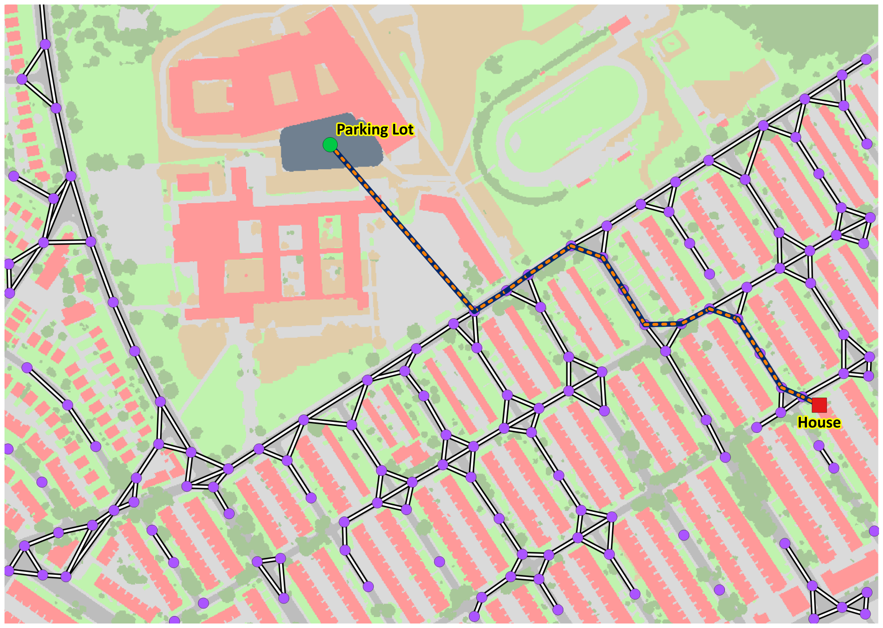


Figure 16. The path found by path-connected search is shown by the dark path that connects the *Parking Lot* node to the *House* node. The road network segments and path-adjacent edges are shown from Figure 13.

with no background image. These figures show the path-connected edge linking the start and finish nodes directly in a similar manner to a distance edge. In fact, the path-connected edge can be thought of as another type of distance edge, but with a graph-topology based distance metric instead of Cartesian distance. Maintaining compatibility with the Star and Disconnected Star search algorithms was a key design consideration in developing the path-connected edge model. Since a path-connected edge can be thought of as “a single edge linking two nodes through some valid path traversal through a network of path-segment nodes,” we can use it in any of the existing GeoGraphy search algorithms like Star and Disconnected Star search.

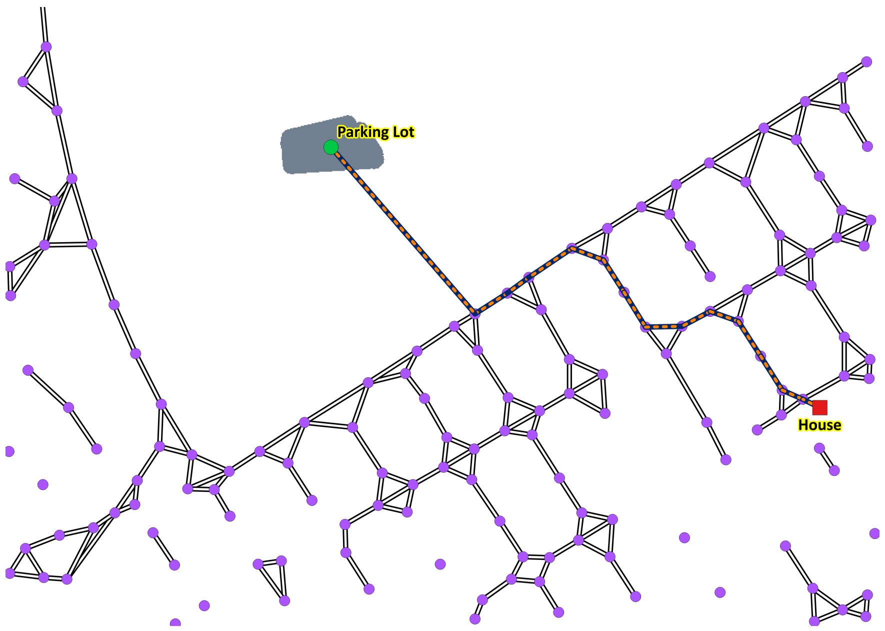


Figure 17. The graph network for the road segments with the path sequence shown connecting the *Parking Lot* node to the *House*. The land cover image is not shown in the background.

Discussion

The path network recovery process is designed to work with the GeoGraphy framework in which a geospatial-temporal semantic graph (GSTSG) is generated from sensor and imagery data. The primary motivation behind this work was based on turning path network nodes into something that can be effectively searched. Prior to this, path networks were difficult to use in searches because they actually represent a collection of overlapping paths. Due to the way GeoGraphy constructs nodes, these overlapping paths get merged into a single graph node.

Our work has focused on separating the node into segments at intersections, repairing broken path segments, and enabling the use of these paths in GeoGraphy's search algorithms. An example of path repair is given in Appendix D [13]. By using the path information in our searches we can answer questions that we were previously unable to address, such as “Is there a path connecting nodes A and B?” (path finding), or “What places can I go to if I start at node A and follow only roads?” (reachability), etc.

The GeoGraphy code has a powerful set of capabilities that has been developed over several years, and one design goal was to leverage as much of the existing capability as possible. Toward that end we used the existing chunking system, which we renamed to *Geometric_Operations* for more generality, as the interface to the partitioning process. The path-connected edge was similarly designed to maximize compatibility with the core GeoGraphy search algorithms by collapsing the path into a single path-connected edge that directly connects our start and finish nodes, enabling its use within the Star, and Disconnected Star, and Connected Component search operations. Fig-

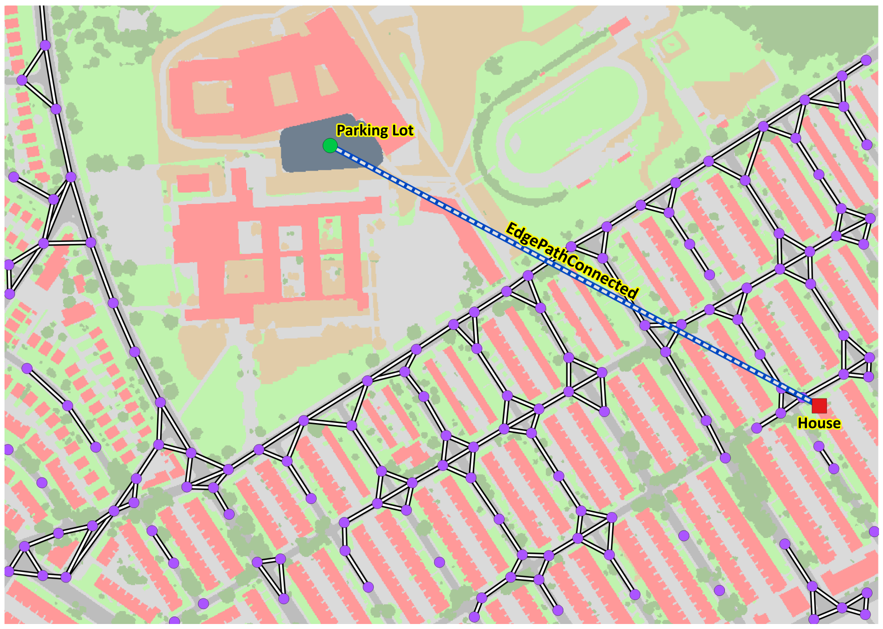


Figure 18. The path-connected edge connecting the *Parking Lot* to the *House*.

ure 15 provides an illustration of the path-connected edge and how it represents the path in relation to the actual path taken through the graph.

Though not a focus of this document, the disconnected star algorithm could be used in conjunction with path-connected edges to find paths between two nodes *A* and *B* with interruptions. For example, suppose we wished to change our "Driving Home" example to allow our driver to cut through parking lots. We could do this by generating path-connected edges between the start node and the finish node, with "Other Paved" nodes allowed as interruptions. This capability will enable powerful path-aware searches to be performed in the future.

Finally, we demonstrated these capabilities using a scenario with a goal of finding the shortest route along roads from work to home. We provided a detailed view of the GeoSearch operations required for the path partitioning and path connected edge search for this problem. In appendix A we provide a brief overview of the full sequence of searches we used for this problem.

Our current implementation separates path partitioning and path repair into separate steps. In future work we hope to better integrate these computations into an integrated path network recovery capability, with a simpler interface for path partitioning, recovery, and search.

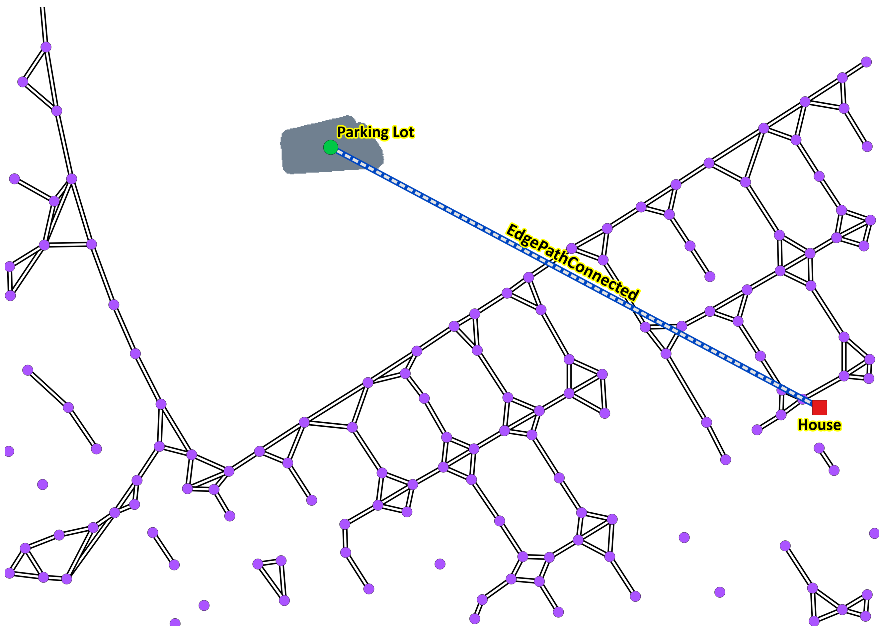


Figure 19. The graph network of road segments with the path-connected edge shown connecting the *Parking Lot* node to the *House*. The land cover image is not shown in the background.

Overview of the “Driving Home” Search

Here we provide a more detailed overview of the setup and sequence of searches used in the “Driving Home” example. This is intended to provide a high level understanding of how the experiment was designed, so that the reader can better understand the full experiment should they be interested.

Constructing the StoredGraph

We construct the StoredGraph for the “Driving Home” example using two data sources. The first one is a land cover image containing the base region data. The second data source is a CSV file containing the point locations of the parking lot we will use as the starting point of the path as well as the location of the destination house.

The land cover image is shown in Figure A.1 This is a standard land cover image showing buildings, roads, and other features. An inspection of the land cover classes in our image reveals that there is no “Parking Lot” class. We will generate one using geometric morphology operations on the “Other Paved” nodes, and the point data in the `Locations.csv` file.

The `Locations.csv` file has the following lines:

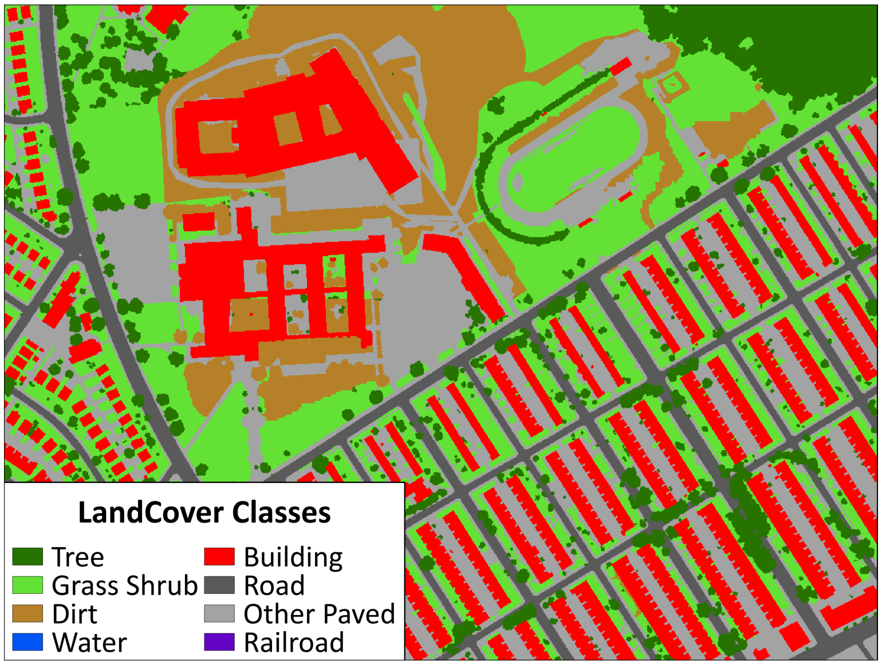


Figure A.1. The land cover image of the scene we used for the "Driving Home" example described in this paper.

Locations.csv

```
Feature Name,Class,Latitude,Longitude
House,Residence,40.040850,-75.035380
Parking Lot,Workplace,40.043674,-75.041389
```

For our example, we use the point data in two different ways. First, we use it to identify the parking lot that our path will begin from. Rather than trying to identify which specific graph node the parking lot is directly from the StoredGraph indices, we can locate it using a point node set in the interior of the parking lot shape, find the node using a GeoSearch, and then write it back as a MatchNode with its role set for easy lookup during our path search. Second, we use the point data to represent the destination of our path. We could use it to identify the specific graph node of the house we are navigating to, but houses are relatively small and we can just use the point itself as the destination node. The result of this will be a path search that will ultimately search for a path between a *region* node to a *point* node.

Parking lots come from the "Other Paved" class, but you might notice that the large regions that are obviously parking lots are also connected by "skinny" segments, typically roads or sidewalks, that connect the large paved regions to one another. Just as intersecting road segments correspond to one large image region that is represented by a single node in the StoredGraph, so do interconnected parking lots, sidewalks, etc. We will correct this and extract our parking lot using geometric operations that shrink then grow the "Other Paved" regions to separate the large block areas from one another.

The other important node type for this example is "Road." We will use the path partitioning search to break the monolithic road nodes into smaller segments that can be used for path search later. These steps are essentially data preparation steps. After they are complete, we can then execute the path-connected search. In the next section we describe the sequence of searches that were used to produce our "Driving Home" result.

Search Sequence

Table A.1. "Driving Home" Sequence of Searches

Search Index	Search Name
1000_Paved_Chunk_10_300	Shrink/Grow "Other Paved" regions.
1010_path_railroad	Apply path partitioning to "Railroad" regions.
1020_path_road	Apply path partitioning to "Road" regions.
1999_Finish_Batch	Checkpoint the StoredGraph after initial sequence.
2010_parking_lot	Identify the *Parking Lot* node using a point location node.
2020_path_connected_search	Find the shortest path from the *Parking Lot* to the *House*.

Table A.1 provides the full list of searches performed by our "Driving Home" example. We typically use a naming convention that provides a four digit search number prefixed to the search name to help us easily maintain the ordering of our search sequence. This is merely a convenience for sorting directory listings, etc. and is not required by the GeoGraphy code.

The first search, *1000_Paved_Chunk_10_300* performs a *chunking* operation that splits the "Other Paved" regions apart from each other by first shrinking the regions by 10 pixels and then growing by 10 pixels. Only chunks of area greater than 300 m^2 will be retained and written to the StoredGraph as MatchNodes.

Next, we run the *1010_path_railroad* search. This search is essentially a no-op since our scene has no railroads, but if it did we would apply the path partitioning code to all railroad nodes present in the scene. This search was included as a check to verify proper operation when no nodes are present to be partitioned.

After this, the *1020_path_road* search runs the path partitioning search on road nodes. This search is discussed in detail in Section 2 and is responsible for generating road segment nodes that will be used later.

1999_Finish_Batch is a standard checkpoint step that is just a no-op. We use these to divide major stages of search sequences for organizational purposes. In deployment environments, searches prior to this standard checkpoint would be run automatically in batch and their results stored in the StoredGraph, providing a semantically rich search environment for the user to begin interactive search.

The *2010_parking_lot* search finds the start node that we will use for the path search. This search uses a Star search to find our parking lot by identifying the paved chunk that was generated in

1000_Paved_Chunk_10_300 that is within 0 m of the *Parking Lot* node from `Locations.csv`. Since this point was located on the interior of the parking lot region, there will be only one matching node from this search. We save the resulting single match to the StoredGraph.

Last, the *2020_path_connected_search* is the search we use to find the shortest path along road network nodes from the *Parking Lot* node, generated in *2010_parking_lot* to the *House* node, that was loaded from `Locations.csv`. This search is described in detail in Section 3.

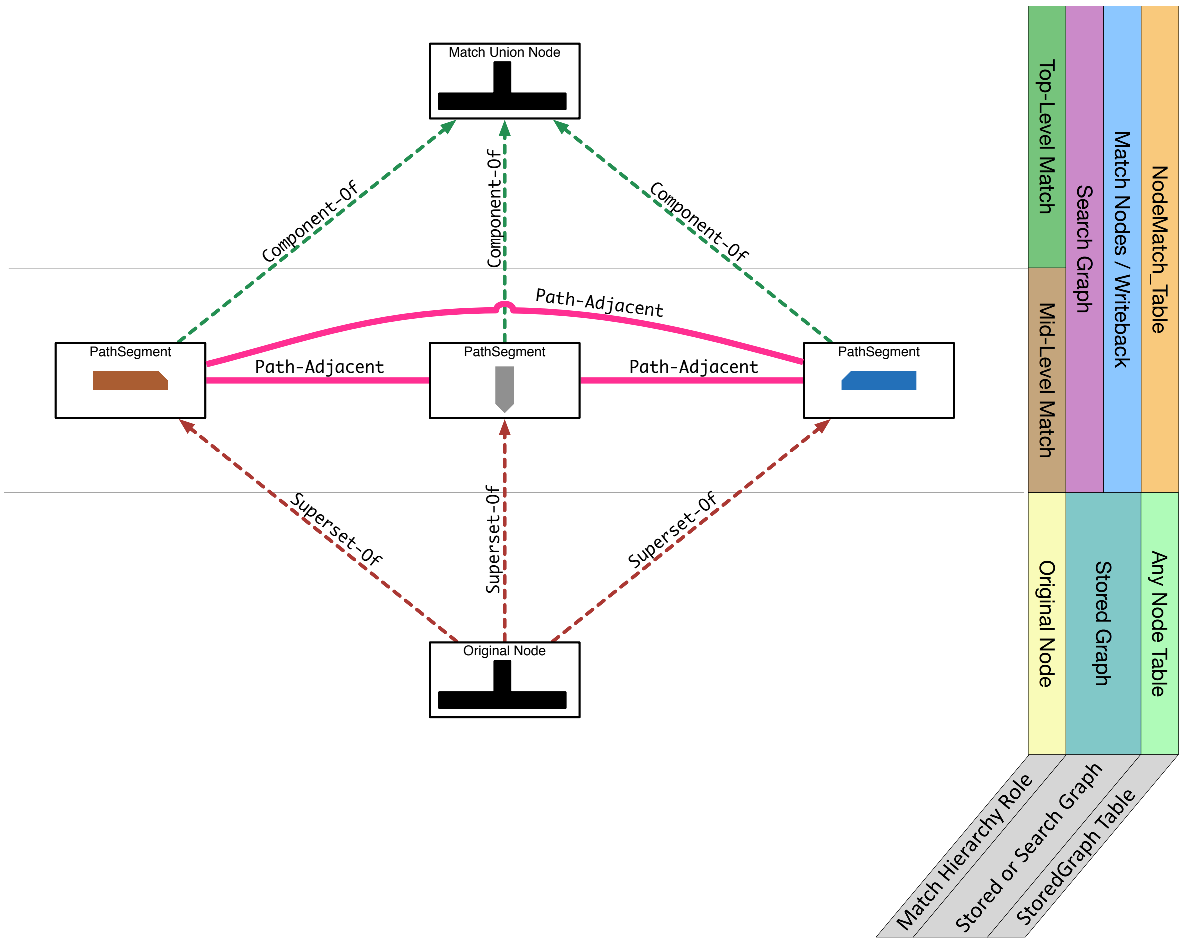


Figure B.2. The match hierarchy of a node that was partitioned by the path-partitioning process. The bar on the right side of the diagram indicates where the different nodes exist in the hierarchy as well as the roles they play in a match hierarchy.

Match Hierarchy

GeoSearch match objects are written back to a StoredGraph in a hierarchical graph structure that allows us to track a given match object back to its original source. This appendix explains the match hierarchy of MatchNodes generated in the path partitioning search.

Figure B.2 illustrates the match hierarchy from our path partitioning search. The graph in the diagram shows the sub-graph representing our match hierarchy, and the labels on the right side of the diagram indicate where these nodes exist and their roles within a match hierarchy. The following sections will provide a brief explanation of each of the three tiers in the match hierarchy.

Original Node

The original node that is to be partitioned is shown in the bottom middle of Figure B.2. This node was present in the StoredGraph before path partitioning began and won't be loaded directly into the SearchGraph during a path partitioning operation, because partitioning occurs during the node construction phase of SearchGraph creation. In this example there are three partitions generated from our original node that we call *mid-level* match nodes.

Mid-Level Match Nodes

Mid-level match nodes are the nodes in a SearchGraph that are used in the GeoSearch. These nodes exist in the SearchGraph, and when they participate in a successful GeoSearch match they are written back into the StoredGraph. Additionally, these nodes are assigned a node role based on the `RoleNodeSpec.csv` file. *Superset-Of* edges are generated to link them to the original StoredGraph nodes they came from. Supserset-of edges are directed edges that always point from the original StoredGraph node to mid-level match nodes and can be read as "*Original node* is a superset of the *mid-level match node*." We should mention that all searches will generate these mid-level match nodes, even when the mid-level match node is identical to the original node. This is because match nodes are given search roles as part of the GeoQuestion specification. For example, a StoredGraph node might be a "Building" but when set in the context of a particular search it may have the role "House." Mid-level match nodes are written to `MatchNode_Table` in a StoredGraph.

Top-Level Match Nodes

At the top of the hierarchy there is a top-level match node. There is exactly one top-level match node for every match found by a GeoSearch. A directed *Component-Of* edge is created from each mid-level match node to their corresponding top-level match node when a match is written back to the StoredGraph. We can read these edges as "A *mid-level* node is a component of the top-level node." Each top-level node stores the shape union of all of its constituent mid-level match nodes and is saved into the StoredGraph in `MatchNode_Table`. Top-level match nodes are assigned the role specified in the `Role.csv` file in the GeoQuestion.

Top-Level match nodes also add a new complication to the concept of a node in a StoredGraph, because their component mid-level nodes might contain both region and point data. MatchNodes take this into account and can store these "hybrid" nodes properly. This is important to note as we often use mid-level and top-level match nodes generated in one search as the input to subsequent searches, which means nodes at any level of this hierarchy can be point, region, or hybrid nodes.

References

[1] Linear skeletons from square cupboards. *Machine Intelligence IV*, pages 403–420, 1969.

[2] Randy C. Brost, William C. McLendon-III, Ojas Parekh, Mark D. Rintoul, David R. Strip, and Diane M. Woodbridge. A computational framework for ontologically storing and analyzing very large overhead image sets. In *3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial-2014)*, November 2014.

[3] Randy C. Brost, David N. Perkins, and Kristina Czuchlewski. Activity analysis with geospatial-temporal semantic graphs. Sandia report (in preparation), Sandia National Laboratories, September 2015.

[4] Sukmoon Chang. Extracting skeletons from distance maps. *International Journal of Computer Science and Network Security (IJCSNS)*, 7(7):213–219, July 2007.

[5] Edmond Chow. *A graph search heuristic for shortest distance paths*. United States Department of Energy, 2005.

[6] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14(3):227 – 248, 1980.

[7] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[8] eCognition. *eCognition Developer 8.7.2 Reference Book*. Trimble, 2012.

[9] R. Fabbri, L. F. Estrozi, and L. Da F. Costa. On voronoi diagrams and medial axes. *Journal of Mathematical Imaging and Vision*, 17:27–40, 2002.

[10] Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast discovery of connection subgraphs. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 118–127, New York, NY, USA, 2004. ACM.

[11] S Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM.

[12] Zicheng Guo and Richard W. Hall. Parallel thinning with two-subiteration algorithms. *Commun. ACM*, 32(3):359–373, March 1989.

[13] William McLendon III and Randy C. Brost. Path network recovery using remote sensing data and geospatial-temporal semantic graphs: Additional appendices (ouo). Sandia report (in preparation), Sandia National Laboratories, September 2015.

[14] S. Krinidis. Fast 2-d distance transformations. *Image Processing, IEEE Transactions on*, 21(4):2178–2186, 2012.

[15] J. O'Neil-Dunne, J.P.M., S.W. MacFaden, and K.C. Pelletier. Incorporating contextual information into object-based image analysis workflows. In *ASPRS 2011 Annual Conference*, May 2011.

[16] J. O'Neil-Dunne, S. MacFaden, A. Royar, M. Reis, R. Dubayah, and A. Swatantran. An object-based approach to statewide land cover mapping. In *ASPRS 2014 Annual Conference*, March 2014.

[17] J. O'Neil-Dunne, K. Pelletier, S. MacFaden, A. Troy, and J. M. Grove. Object-based high-resolution land-cover mapping: Operational considerations. In *2009 17th International Conference On Geoinformatics*, pages 1–6. IEEE, 2009.

[18] Jarlath P.M. O'Neil-Dunne, Sean W. MacFaden, Anna R. Royar, and Keith C. Pelletier. An object-based system for lidar data fusion and feature extraction. In *Geocarto International*, volume 28, pages 227–242. Taylor and Francis Group, 2012.

[19] QGIS Development Team. *QGIS Geographic Information System*. Open Source Geospatial Foundation, 2009.

[20] Jean-Paul Watson, David R. Strip, William C. McLendon III, Ojas Parekh, Carl Diegert, Shawn Martin, and Mark D. Rintoul. Encoding and analyzing aerial imagery using geospatial semantic graphs. Sandia Report SAND2014-1405, February 2014.

[21] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, March 1984.

DISTRIBUTION:

1	MS 0401	Kurt Larson, 05563
1	MS 0519	Steve Castillo, 05340
1	MS 0519	Michelle Carroll, 05346
1	MS 0519	James Morrow, 05346
1	MS 0519	Kristina Czuchlewski, 05346
1	MS 0974	David Perkins, 05541
1	MS 0974	Jamie Coram, 05544
1	MS 1326	Randy Brost, 01462
1	MS 1326	Mark D. Rintoul, 01464
1	MS 1326	William C. McLendon III, 01462
1	MS 1327	John Wagner, 01462
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories