Lab Report

Title: Lab1

Notice: Dr. Bryan Runck

Author: Rob Hendrickson

Date: 11/02/2022

Repository: https://github.com/RwHendrickson/GIS5571/tree/main/Lab03/Part_1

Time Spent: 15 hours

Abstract

In this lab, I performed sensitivity analysis on walk-ability cost surfaces for the area around Minnesota's Whitewater State Park. The problem statement section provides contextual information and requirements for analysis. The input data section will describe the different datasets acquired in this exercise. The methods section includes detailed visual and textual descriptions of the workflow conducted. The results section will present some visualizations of the surfaces and cost paths that were computed. This report concludes with a discussion on the lessons learned and future directions of this work.

1 Problem Statement

The client of this project is a fictional character named Dory who lives on a farm near the borders of Olmsted, Wabasha, and Winona counties. She enjoys walking to the North Picnic Area of Whitewater State Park from time to time. Her journey is often quite enjoyable, however, it can also be quite treacherous if she chooses the wrong route at the wrong time of year. Her largest concerns are muddy farm fields in the spring and water bodies if there isn't a bridge (or she has her waders). Otherwise, she just prefers the most gradual slope. On the following page is a map of her journey and a table of the requirements to analyze her best routes.

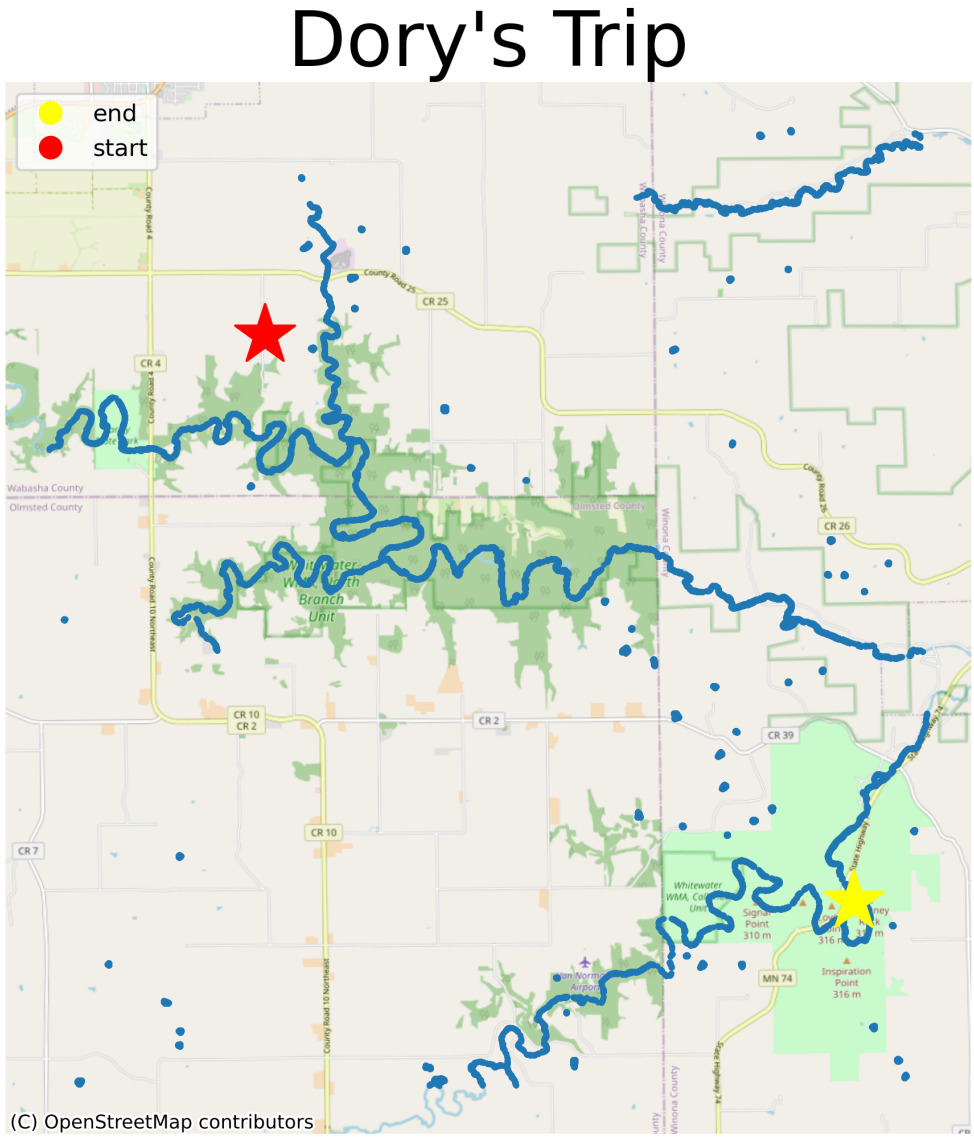

Figure 1: A map of the study area with trip origin, destination, and water highlighted.

	Requirement	Defined As	Data (Spatial)	Data (Attribute)	Dataset	Preparation
1	Data Acquisition	Request Light Detection and Ranging (LiDAR), land use, road, and waterway data	Point clouds, polygons, polylines	Elevation, MLCCS Code	Minnesota Dept. of Natural Resources (LiDAR) Minnesota Land Cover Classification System (MLCCS) Minnesota Dept. of Transportation (MnDoT) Minnesota Dept. of Natural Resources (Public Waters)	Navigated API trees
2	Process Data	Clip to extent, merge LiDAR tiles, select relevant land classifications, erase roads from waterways	Point Cloud, polygons, polylines	Elevation, Classification Code, Watercourse Name	LiDAR Point Clouds, MLCCS Agricultural Land, Public Watercourses minus Public Roadways	Decompress LiDAR files, Study MLCCS coding scheme, Explore Public Water Basin
3	Synthesize data	Make all information interoperable	Rasters	Max Elevation (Centimeter) is_field (binary) is_water_no_bridge (binary)	Rasterized elevation Rasterized Fields Rasterized Waterways without bridges	Verify processed data
4	Calculate Elevation Cost	Compute slope degrees from elevation	Raster	Slope Degree	Rasterized Slope Degree	Compute the gradient of the elevation raster
5	Transform Slope Degree	Logarithmic transform of left-tailed slope degree values	Raster	Transformed Slope Degree	Rasterized Log Slope Degree	Explore distribution of elevation and slope
6	Normalize	Ensure that all values are in a similar range with appropriate sign	Rasters	Normalized log slope degree (float) is_water_no_bridge (binary) is_field (binary)	Rasterized Log-norm slope degree is_water_no_bridge is_field	Explored distribution of values
7	Simulate Cost Surface Creation	Interate through linear combinations between the 3 rasters with weights that sum up to 1	Raster	Walk-ability Cost	Cost Surface	
8	Cost Path Analysis	Compute Least Cost Paths for each cost surface	Rasters		Least Cost Path	Study least cost path analysis
9	Uncertainty Analysis	Compare cost surfaces and least cost paths	Rasters and Polylines	Walk-ability Cost	Cost Surface, Least Cost Path	Vectorize least cost paths

Table 1: Project requirements

2 Input Data

The LiDAR data acquired from the Minnesota Department of Natural Resources (MnDNR) were 20 LiDAR tiles from Wabasha, Winona, and Olmsted counties. The full list of tiles is in figure 2 and were determined using each county's respective file title, `tile_index_map.pdf`. These tiles contain geographic information, classification (water, ground, bridge, vegetation, etc), and elevation.

The polygons acquired from Minnesota Land Cover Classification System (MLCCS) contained classifications based on the MLCCS coding scheme.

Another dataset from MnDNR was the public watercourses and basins of Minnesota. This was combined with the public roads from the Minnesota Department of Transportation (MnDoT) to create the water cost layers.

Their API links can be found in table 2.

```
[('winona', '4342-28-62'),
 ('winona', '4342-28-63'),
 ('winona', '4342-29-62'),
 ('winona', '4342-29-63'),
 ('winona', '4342-30-62'),
 ('winona', '4342-30-63'),
 ('winona', '4342-31-62'),
 ('winona', '4342-31-63'),
 ('wabasha', '4342-28-59'),
 ('wabasha', '4342-28-60'),
 ('wabasha', '4342-28-61'),
 ('olmsted', '4342-29-59'),
 ('olmsted', '4342-29-60'),
 ('olmsted', '4342-29-61'),
 ('olmsted', '4342-30-59'),
 ('olmsted', '4342-30-60'),
 ('olmsted', '4342-30-61'),
 ('olmsted', '4342-31-59'),
 ('olmsted', '4342-31-60'),
 ('olmsted', '4342-31-61')]
```

Figure 2: Full list of LiDAR tiles in the format (county, tile id)

	Title	**Purpose in Analysis**	**Download Link**
1	MnDNR LiDAR Tiles	Determining slope	https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/
2	MLCCS Land Cover Classifications	Determining location of fields	https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dnr/biota_landcover_mlccs/shp_biota_landcover_mlccs.zip
3	MnDNR Public Waters	Determining location of water	https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dnr/water_mn_public_waters/shp_water_mn_public_waters.zip
4	MnDoT Annual Average Daily Traffic (AADT)	Determining location of bridges	https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dot/trans_aadt_traffic_segments/shp_trans_aadt_traffic_segments.zip

Table 2: Data Sources

3 Methods

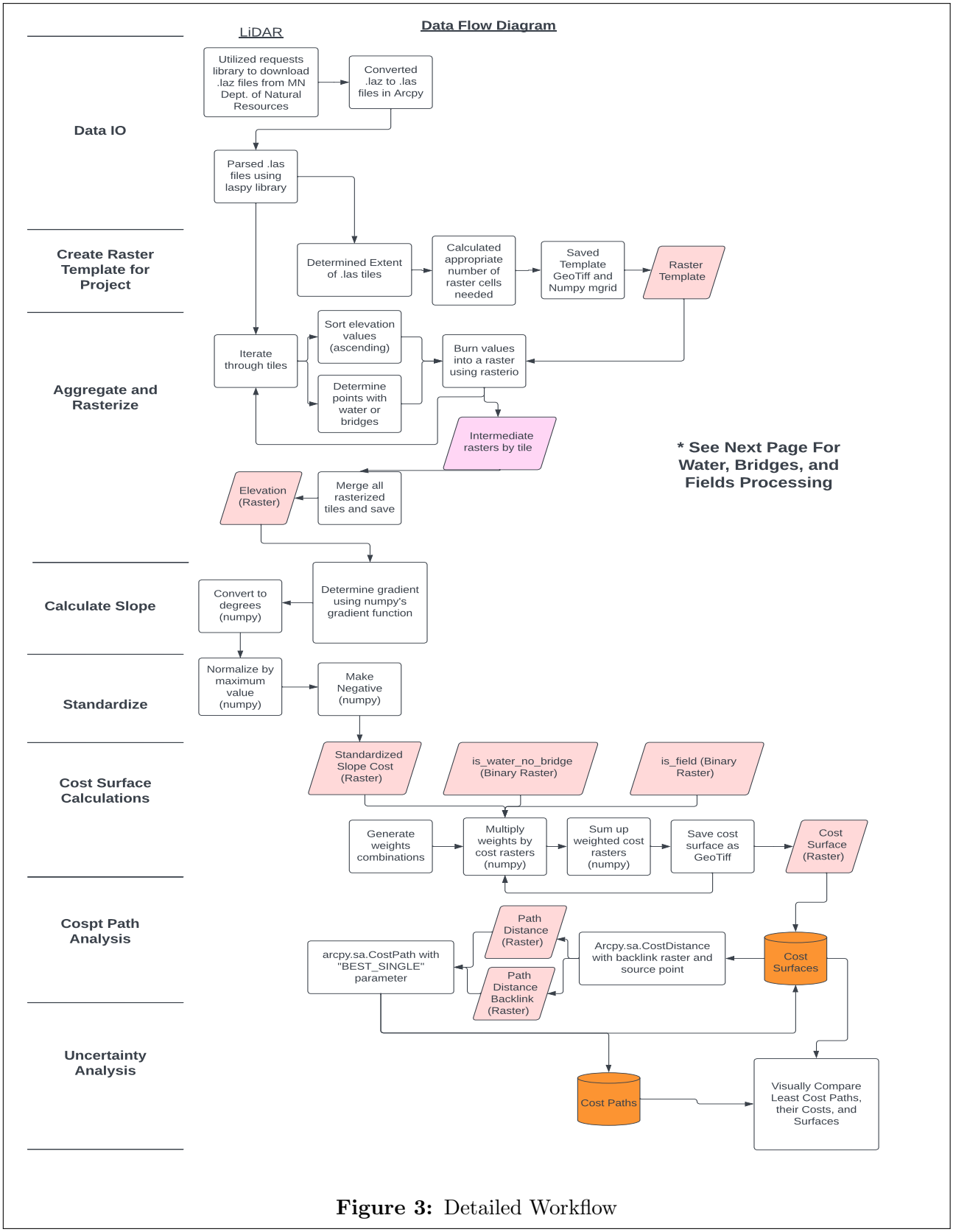

Figure 3: Detailed Workflow

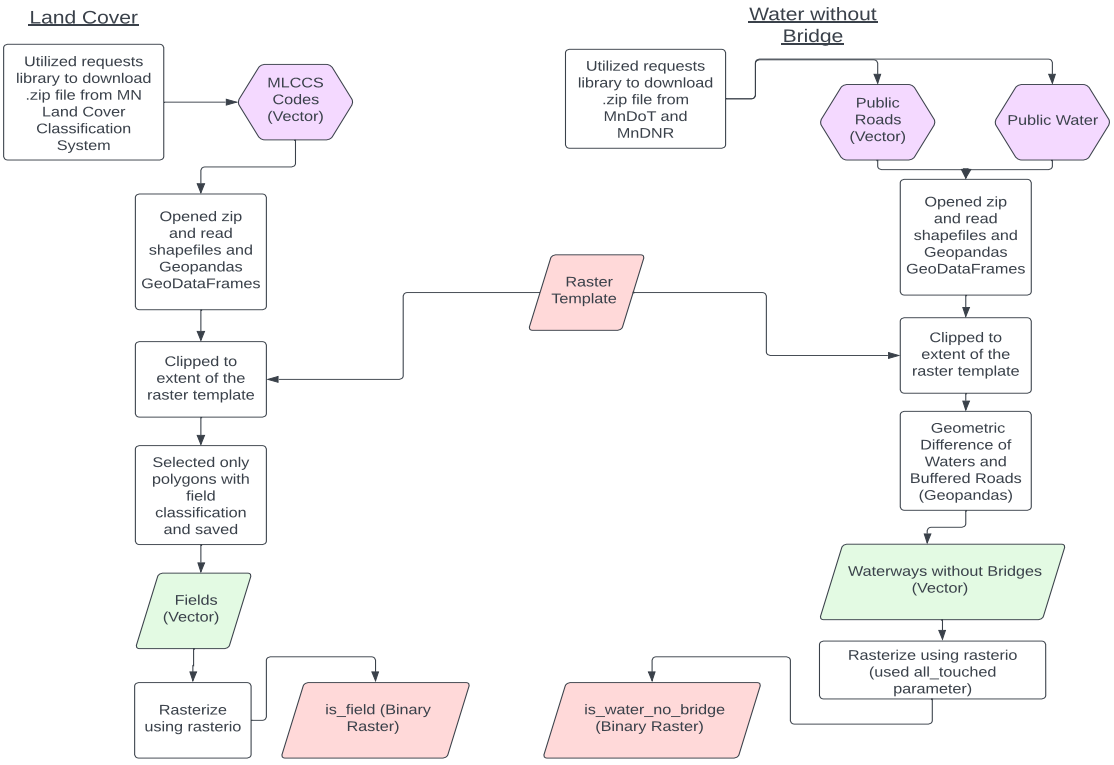

Figure 4: Data IO and Processing of Fields and Water

3.1 Data Input/Output

3.1.1 MnDNR (LiDaR)

To diagnose which LiDAR tiles were needed from MnDNR, the tile index maps were consulted from each county. An example of this map can be found here: https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/winona/tile_index_map.pdf. This list of tiles was iterated over and their respective `.laz` files were requested and written to disk. These files were decompressed into `.las` files utilizing Arcpy's ConvertLas function. Code to accomplish these tasks is provided in figures 5 and 6.

```
# Download Data

downloaded = True

if not(downloaded):
    # Takes like 5 minutes? Maybe?
    # Total size is 458mb

    base_url = 'https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/'

    for tile_stuff in tile_info:

        county = tile_stuff[0]
        tile_index = tile_stuff[1]

        tile_name = tile_index + '.laz'
        tile_url = base_url + county + '/laz/' + tile_name

        savepath = os.path.join('1_raw_data', 'LazTiles', tile_name)

        if not(os.path.exists(savepath)):

            response = requests.request("GET", tile_url) # Get request

            # Save

            with open(savepath, "wb") as file:
                file.write(response.content)
```

Figure 5: Iteratively interfacing with MnDNR's API

```python
# Convert Laz to Las

converted_to_las = True

if not(converted_to_las):

    # Arcpy Stuff

    import arcpy # Arcpy

    # Set Working Directory (Arcpy)

    arcpy.env.workspace = os.path.join(os.getcwd(), 'Arc1_Lab02_part2.gdb')

    # Convert to las

    arcpy.conversion.ConvertLas('Part_2_LazTiles', os.path.join('1_raw_data','LasTiles'),
                                '1.4', 7, 'NO_COMPRESSION',
                                define_coordinate_system = 'ALL_FILES',
                                in_coordinate_system = arcpy.SpatialReference(26915))
```

Figure 6: Iteratively converting `.laz` files to `.las` files

Once the files were in a `.las` format, they could be accessed with the Python library, `laspy`. The library claims to work with `.laz` files but it didn't work with my version. The extent of all the tiles was ascertained with the code in figure 7. This information was used to create a template raster for analysis (see section 3.2). It should be noted that the `.las` units are in centimeters.

```python
# Find extent of all the tiles

minx = np.inf
miny = np.inf
maxx = np.NINF
maxy = np.NINF

for tile_name in tile_names:

    path = os.path.join('1_raw_data', 'LasTiles', tile_name)

    las = laspy.read(path)

    # Check extent of tile

    if min(las.X) < minx:
        minx = min(las.X)
    if min(las.Y) < miny:
        miny = min(las.Y)
    if max(las.X) > maxx:
        maxx = max(las.X)
    if max(las.Y) > maxy:
        maxy = max(las.Y)
```

Figure 7: Getting the extent of all `.las` tiles

3.1.2 MLCCS

Land cover from MLCCS was accessed through the Minnesota Geospatial Commons's API. This link can be found in table 2. A zip file was downloaded and the shapefile within titled, `landcover_minnesota_land_cover_classification_system.shp`, was read into Geopandas. Here it was clipped to the extent of the template raster (see section 3.2) and selected for the classification, 22 (agricultural land). This selected GeoDataFrame was saved in a `.geojson` format. The code that clips, selects, and saves is provided in figure 8.

```
# Clip to extent

# minx = 56495713 miny = 487566062 maxx = 57761522 maxy = 488967939
# These are in centimeters!!!

# Load template raster bounds

# Get bounds from template
rst = rasterio.open('template.tif') # Open template
bounds = rst.bounds
rst.close()

minx, maxy, maxx, miny = bounds

# Create a custom shapely polygon
extent = Polygon([(minx,miny), (minx, maxy), (maxx, maxy), (maxx, miny), (minx,miny)])
extent_gdf = gpd.GeoDataFrame([1], geometry=[extent], crs='EPSG:26915')

# Clip

landcover_clipped = landcover.clip(extent_gdf)

# Save Fields (22 classification)

landcover_clipped[landcover_clipped.CARTO == 22].to_file(os.path.join('1_raw_data', 'fields.geojson'))
```

Figure 8: Clipping, selecting, and saving the land cover dataset.

3.1.3 Water, No Bridge

Water and road spatial information was obtained from the Minnesota Geospatial Commons' API much in the same way as the MLCCS data. These links can be found in table 2. After clipping to the extent of the study, the roads were buffered 30 meters and the geometric difference was taken between the water and the buffered roads. The resulting GeoDataFrame was saved in a `.geojson` format. The code that performs the geometric difference is provided in figure 9, and the saved file is visualized in figure 10.

```
### Clip bridges from water

# Find intersections (bridges)

bridges = course_clipped.geometry.unary_union.intersection(road_clipped.geometry.unary_union)

# Create 30 meter buffer around bridge points (Crossings)

buffed_bridges = bridges.buffer(30)

# Convert Shapely multipolygon to geoseries

buffed_bridges_gdf = gpd.GeoDataFrame(geometry = gpd.GeoSeries(buffed_bridges, crs = 'EPSG:26915'))

# Erase them out of waterlayer

water_no_bridge = course_clipped.overlay(buffed_bridges_gdf, how='difference')
```

Figure 9: Taking the geometric difference.

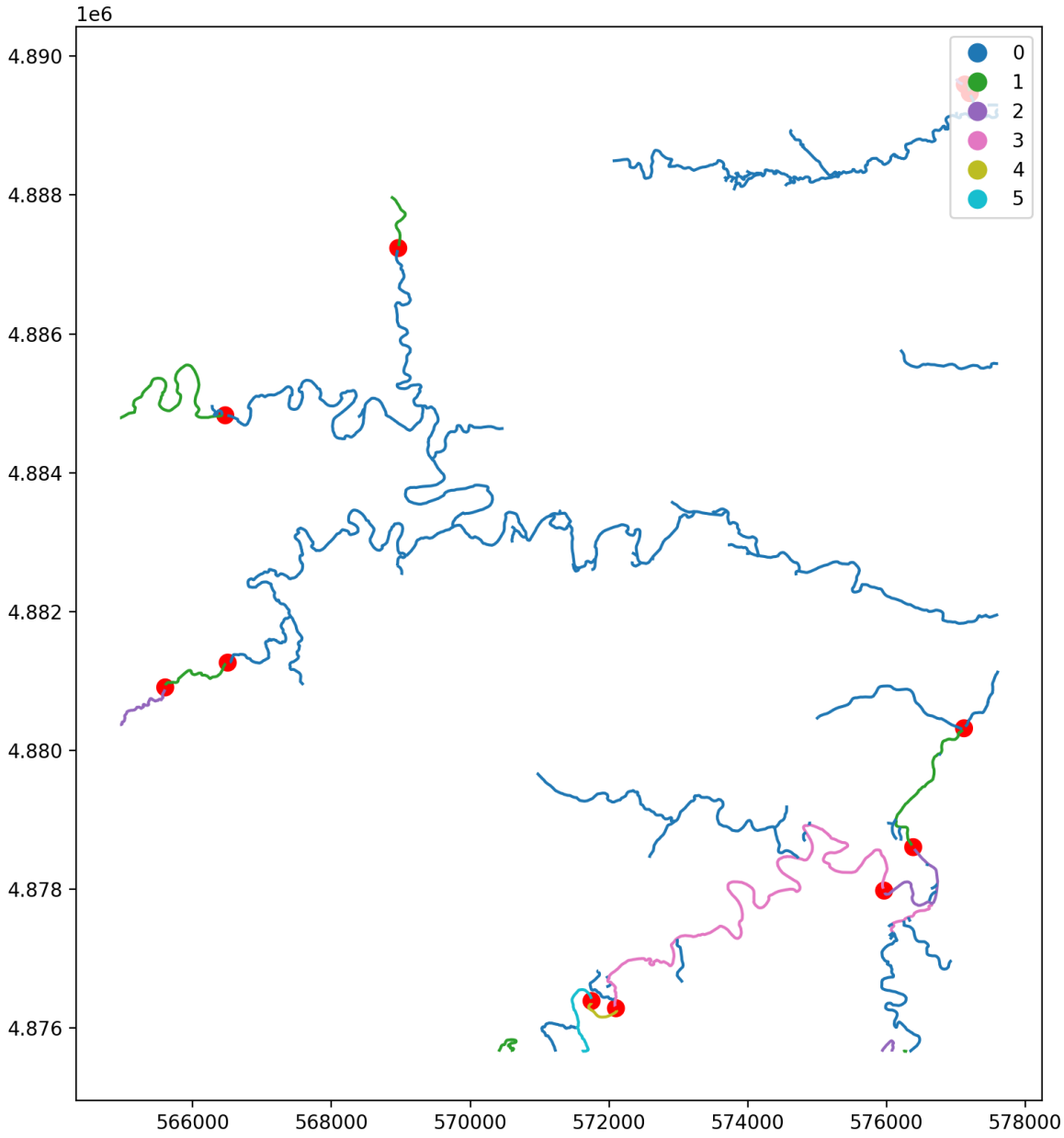

Figure 10: A map highlighting the bridges and segment indices of the split waterways.

3.2 Creating Raster Template

Upon identifying the extent of the `.las` files, the height and width of the study was computed. These were used to calculate dimensions of a regular grid of 10 meter resolution for the space. This grid was saved in both a Geotiff raster format and numpy mgrid. The code that executes these operations is in figures 10 - 14.

```
# Find the width and height (in meters)

extent = minx_meters, miny_meters, maxx_meters, maxy_meters = (minx/100, miny/100, maxx/100, maxy/100)

height = extent[3] - extent[1]
width = extent[2] - extent[0]

print('height = ', height/1000, 'kilometers, width = ', width/1000, 'kilometers')
```

```
height =  14.018769999999552 kilometers, width =  12.658089999999968 kilometers
```

Figure 11: Calculating width and height of study extent.

```
# Create raster

resolution = 10 # Want maybe 10 x 10m meter cells

x_correction = resolution - np.mod(width, resolution) # It won't be perfect unless we add these
y_correction = resolution - np.mod(height, resolution)

# Split correction amongst min & maxs

new_minx = minx_meters - x_correction/2
new_maxx = maxx_meters + x_correction/2
new_miny = miny_meters - y_correction/2
new_maxy = maxy_meters + y_correction/2

# New heights

new_height = new_maxy - new_miny
new_width = new_maxx - new_minx

print('Corrected Height (for 10m resolution) = ', new_height, 'width = ', new_width)

x_cells = new_width/resolution
y_cells = new_height/resolution

print('x_cells = ', x_cells, '\ny_cells = ', y_cells)
```

```
Corrected Height (for 10m resolution) =  14020.0 width =  12660.0
x_cells =  1266.0
y_cells =  1402.0
```

Figure 12: Computing number of cells in each dimension for raster.

```
# Definitions

def save_geotiff(array, name, crs, resolution, minx, miny):
    '''Saves a numpy array into a geotiff.

    Give name as a string
    crs as int, resolution as int
    minx and miny both as floats
    '''
    transform = Affine.translation(minx - resolution / 2, miny - resolution / 2\
                                   ) * Affine.scale(resolution, resolution)

    with rasterio.open(
        os.path.join(".", name + '.tif'),
        mode="w",
        driver="GTiff",
        height=array.shape[1],
        width=array.shape[0],
        count=1,
        dtype= 'float64',
        crs= rasterio.crs.CRS.from_epsg(crs),
        transform=transform,
    ) as new_dataset:
        new_dataset.write(array, 1)
```

Figure 13: A function for saving a geotiff file.

```
# Save Template as a GeoTiff

array_temp = np.empty([int(x_cells), int(y_cells)])

save_geotiff(array_temp, 'template', 26915, resolution, new_minx, new_miny)

# Save Template as a Numpy Grid (for plotting)
# Must mannually input cells from above.......

raster = np.mgrid[new_minx:new_maxx:1266j,
                  new_miny:new_maxy:1402j]

np.save('template', raster)
```

Figure 14: Saving the template raster.

3.3 Aggregate and Rasterize

3.3.1 Elevation

To get a full elevation raster, the `.las` files were iterated over. Each was read using the `laspy` library and rasterized using Rasterio's `features.rasterize` function. This function does not have an average values option when merging points into a cell, so their elevations were sorted in ascending order and the merge algebra, 'REPLACE' was used. This effectively burned the largest `.las` point value for each cell into the template raster's format and saved it as a geotiff. Once complete, all intermediate rasters were summed up and overlapping cells were averaged. Code that accomplishes these tasks are in figures 15 and 16. This final raster was saved as a geotiff called, `full_elevation.tif`.

```
# aggregate points into raster
# Takes about 15 minutes

# This version had overlap between the tiles...

for i, tile_name in enumerate(tile_names):

    path = os.path.join('1_raw_data', 'LasTiles', tile_name)

    las = laspy.read(path)

    las_geoms = gpd.points_from_xy(las.X/100, las.Y/100) # Spatialize
    Zs = las.Z # Elevations

    sort_indices = np.argsort(Zs) # Sort ascending

    out_fn = os.path.join('tile_elevations', tile_name[:-4] + '_elevation.tif') # Savepath

    with rasterio.open(out_fn, 'w+', **meta) as out: # Burn features into raster
        out_arr = out.read(1)


        # this is where we create a generator of geom, value pairs to use in rasterizing
        shapes = ((geom,value) for geom, value in zip(las_geoms[sort_indices], Zs[sort_indices]))

        burned = features.rasterize(shapes=shapes,
                                    fill=0,
                                    out=out_arr,
                                    transform=out.transform)#,
                                    # merge_alg=rasterio.enums.MergeAlg.add) # We're not adding them, we're replacing
                                    # This is why we sorted earlier, consistently using the highest value

        out.write_band(1, burned)
```

Figure 15: Iteratively rasterizing the `.las` files.

```
# Load those rasters!

elevs = np.empty([len(tile_names)], dtype = object) # storage for each tile

for i, tile_name in enumerate(tile_names):

    out_fn = os.path.join('tile_elevations', tile_name[:-4] + '_elevation.tif')

    rast = rasterio.open(out_fn) # Open

    elevs[i] = rast.read(1) # Save band

    rast.close() # Close
```

```
# Add them up to get elevation raster

full_elev = np.zeros(elevs[0].shape)

for elev in elevs:

    # Do any cells overlap?

    filled = full_elev > 1
    to_fill = elev > 1


    overlaps = np.logical_and(filled, to_fill)

    # Take average if they overlap

    full_elev[overlaps] = (full_elev[overlaps] + elev[overlaps])/2

    # Otherwise

    full_elev[np.invert(overlaps)] += elev[np.invert(overlaps)]
```

Figure 16: Merging intermediate elevation rasters.

3.3.2 Fields and Water, No Bridge

The fields geojson was loaded as a Geopandas Geodataframes and rasterized using the same process as 3.3.1 (without the iteration) and saved as `rasterized_fields.tif`.

The water, no bridge geojson was also rasterized in a similar manner to Fields and Elevation, however, the parameter, `all_touched` was set to True to close of diagonal water jumps in the cost path analysis.

Originally, the LiDAR data was used to determine water. The `laspy.read()` objects have a classification attribute which yields a numpy array of each point's classification. 2 indicated ground, 9 indicated water, and 14 indicated bridge decks. There is also 8 (model keypoint) and 12 (overlapping points) which could be utilized in future projects. No bridge decks were identified in any of the MnDNR's `.las` files so this route was abandoned.

3.4 Transform and Normalize Costs

Each of the final rasters, `full_elevation.tif`, `water_no_bridge.tif`, and `rasterized_fields.ti` were transformed and/or normalized to align with Dory's preferences. All magnitudes were between 0 and 1 and higher values indicated a higher cost. They were saved in the same format as the template using the function in figure 17.

```python
def Save_Geotiff_to_template(array, template_path, save_path):
    '''Saves a numpy array into a geotiff with the same CRS as the template.
    '''

    # Get metadata from template
    rst = rasterio.open(template_path) # Open template
    meta = rst.meta.copy() # Copy template metadata
    # meta.update(compress='lzw') # Good for integers/categorical rasters
    rst.close()

    with rasterio.open(save_path, 'w+', **meta) as out: # Burn features into raster
        out.write_band(1, array)
```

Figure 17: A function to save an array to a template raster.

3.4.1 Slope

To calculate slope, first the gradient was calculated using Numpy's gradient function on the elevation (in meters) and then this was converted into slope degrees (see code in figure 18). The distribution of values was found to be heavily tailed (see figure 19) so it was transformed using log_{10} which produced a multimodal distribution. The logged slope values were further transformed by shifting values by the minimum value. Finally they were normalized by the maximum value to achieve the "standardized" slope cost surface (code in figure 20, distribution in figure 21).

```python
# Get the slope for the elevation...

# https://livebook.manning.com/book/geoprocessing-with-python/chapter-11/115
# https://gis.stackexchange.com/questions/361837/calculating-slope-of-numpy-array-using-gdal-demprocessing

# This is like the derivative AKA Gradient

elevation = bands[0]/100 # Elevation is also in centimeters
elevation[elevation == 0] = None # Remove any zeros, those are the "no data"
resolution = 10

px, py = np.gradient(elevation, resolution)

slope = np.sqrt(px ** 2 + py ** 2)

slope_deg = np.degrees(np.arctan(slope))
```

Figure 18: Computing slope degrees.

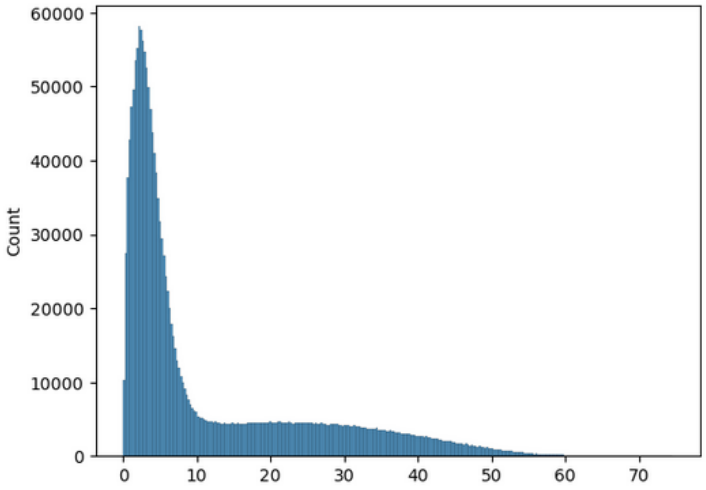

Figure 19: Distribution of slope degree values.

```python
log_slope = np.log(slope_deg)

# I'm going to standardize this by saying:

non_nan = np.invert(np.isnan(log_slope))
non_neg_inf = np.invert(np.isneginf(log_slope))

non_nan_neg_inf = np.logical_and(non_nan,non_neg_inf)

max_ = log_slope[non_nan_neg_inf].max()
min_ = log_slope[non_nan_neg_inf].min()

shifted_log_slope = log_slope - min_ # Shifting so only positive values

std_log_slope_deg = (shifted_log_slope/(max_ - min_)) # Normalizing to range

# Slope of zero (where slope_deg = 0) should be no cost (not infinity)

std_log_slope_deg[slope_deg == 0] = 0

sns.histplot(std_log_slope_deg.flatten())
```

Figure 20: Transforming and normalizing slope degree.

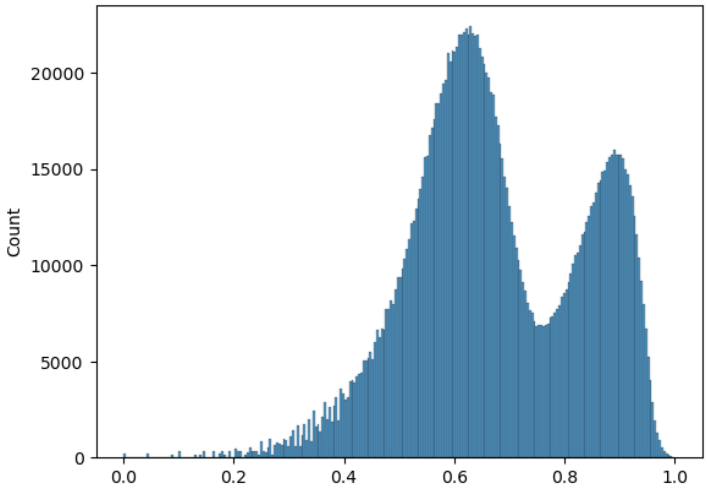

Figure 21: Final distribution of transformed and normalized slope.

3.4.2 Water and Fields

The water and fields binary rasters were ensured to be 1 (if field or water) and 0 otherwise.

3.5 Cost Surface Computation

All weights between costs were standardized to be between 0 and 1. Weights combinations (by increments of 0.1) were generated using code in figure 22. These weights combinations were iterated through and multiplied by the respective cost rasters. The resulting summation of the weighted rasters was saved with the title format `__S-__W-__F-Cost_Surface.tif` where the blanks coincide with the percentage that raster was weighted (S = Slope, W = Water, F = Fields). This code can be seen in figure 23.

```
# Constraints
# S_weight + W_weight + F_weight = 1
# 0 < S_w, W_w, F_w < 1

potential_values = np.linspace(0, 1, 11)

weights_tests = []

for S_w in potential_values:
    for W_w in potential_values:
        for F_w in potential_values:
            if int((S_w + W_w + F_w)*100) == 100:
                weights_tests += [[S_w, W_w, F_w]]
```

Figure 22: Creating weights combinations for cost surface creation.

```
weights_tests = np.array(weights_tests)

for weights in weights_tests:

    title = ''

    cost_surface = np.zeros(rasts[0].shape)

    for i, w in enumerate(weights):

        cost_surface += w * rasts[i]

        title += str(int(w*100)) + rast_names[i][0].upper() + '-'

    title += 'Cost_Surface'

    savepath = os.path.join('5_weights_tests', title + '.tif')

    Save_Geotiff_to_template(cost_surface, 'template.tif', savepath)
```

Figure 23: Iterating through weights combinations and saving resulting cost surfaces.

3.6 Cost Path Analysis

First, features of start and end coordinates were created. In Arcpy the spatial analyst functions `CostDistance()` and `CostPath()` were performed on each cost surface, excluding those without slope as these could not lead to a single best cost path. The necessary parameters and code to perform these steps is given in figure 24.

```
for i, filename in enumerate(file_names):

    if filename[0] != '0': # We don't want any of the costpaths without slope (doesn't work)

        weightnames = filename[:-17] # Weights name (layer name without '-Cost_Surface.tif')

        # Import cost surface

        cs_path = os.path.join(os.getcwd(), '5_weights_tests', filename)
        in_raster = arcpy.Raster(cs_path)

        # Get Path Distance and backlink rasters
        out_distance_raster = arcpy.sa.CostDistance("DoryTrip_start", in_cost_raster = in_raster,
                                                    out_backlink_raster = "PathdistBacklink");
        out_distance_raster.save("Pathdist")

        # Get costpath

        costpath_savename = os.path.join(os.getcwd(), '6_costpaths', weightnames + '_CostPath.tif')

        least_cost_path = arcpy.sa.CostPath("DoryTrip_end",
                                        'Pathdist',
                                        "PathdistBacklink",
                                        "BEST_SINGLE");least_cost_path.save(costpath_savename)

        # Make sure these aren't used again by accident
        arcpy.Delete_management("Pathdist")
        arcpy.Delete_management("PathdistBacklink")
```

Figure 24: Code to iteratively calculate least cost paths on a cost surface.

3.7 Uncertainty Analysis

Select cost surfaces and paths were visualized using matplotlib's pcolormesh plot to understand how varying the weights effected the cost surface. An example function that creates this type of visualization can be found in figure 25. There was an addendum to this code which calculates the cost of a path, vectorizes it, and plots it as well (see figure 26).

```
def six_plot(rast_names, suptitle):

    # Load necessary data

    raster = np.load('template.npy')

    trip = gpd.read_file('DoryTrip.geojson').to_crs('EPSG:26915')

    f, axs = plt.subplots(2,3, figsize = (24,16))

    norm = mpl.colors.Normalize(vmin=-1, vmax=0) # Norm for coloration

    for i, ax in enumerate(axs.flatten()):

        path = os.path.join('5_weights_tests', rast_names[i])

        # Load that raster

        rast = rasterio.open(path) # Open

        values = rast.read(1) # Save band

        rast.close() # Close

        # Plot it

        art = ax.pcolormesh(raster[0], raster[1], values.T, norm = norm, shading='auto', cmap = 'magma')
        title = rast_names[i][:-17]
        ax.set_title(title, fontsize = 16) # Add title
        ax.set_axis_off()

        trip.plot(categorical = True, cmap = 'Dark2',
            # column = 'type', cmap = 'black',
                  # legend = True,
                  marker = '*', markersize = 200, ax = ax)



    f.colorbar(art, ax=axs[:, :], location='right', shrink=0.85)
    f.suptitle(suptitle, fontsize = 24)
    # f.tight_layout()
    f.savefig(suptitle+'.png', dpi = 300)
```

Figure 25: Function to plot six cost surfaces.

```
if with_path:

    if title[0] != '0': # We don't have paths if no slope...

        # Load that raster

        path = os.path.join(os.getcwd(), '6_costpaths', title + '_CostPath.tif')

        rast = rasterio.open(path) # Open

        costpath = rast.read(1) # Save band

        rast.close() # Close

        # Get path coords & Cost

        is_path = np.argwhere(np.rot90(costpath, k=3)==3) # Weird save in Arcpro... Also the numpy mgrid is we
        x_coords = raster[0].T[0][[x[0] for x in is_path]].flatten()
        y_coords = raster[1][0][[x[1] for x in is_path]].flatten()
        #zip the coordinates into a point object and convert to a GeoData Frame
        geom = LineString([Point(xy) for xy in zip(x_coords, y_coords)])
        path_df = gpd.GeoDataFrame(geometry=[geom], crs = 'EPSG:26915')

        cost = values[~np.isnan(values)].T[is_path].sum()

        # Plot
        path_df.plot(ax = ax)
        # ax.pcolormesh(raster[0], raster[1], np.rot90(costpath), shading='auto', cmap = 'Greys', alpha = .3)#

        title = title + '\nPath Cost = ' + str(cost)
```

Figure 26: This code vectorizes a cost path, calculates its cost, and plots it

4 Results

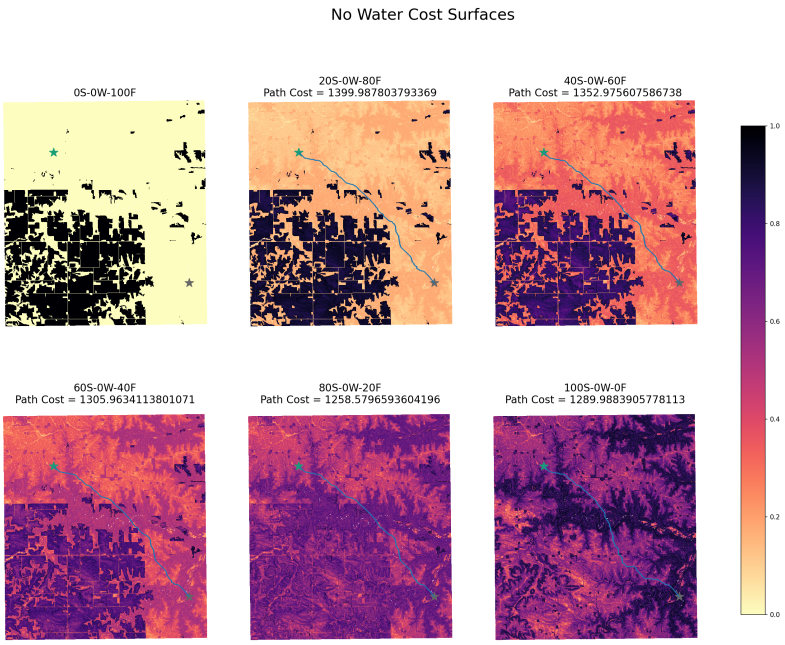

Figure 27: Cost Surfaces with no weight on water cost.

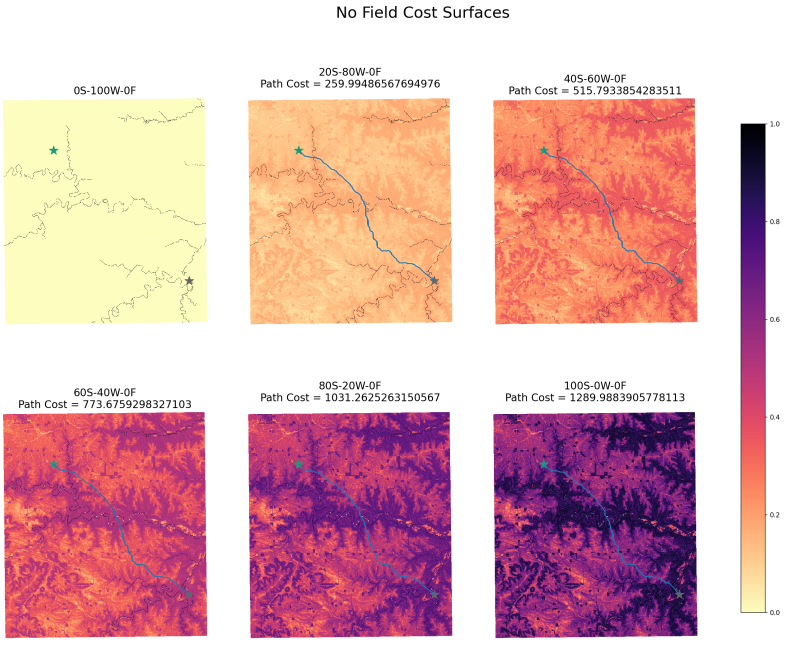

Figure 28: Cost Surfaces with no weight on field cost.

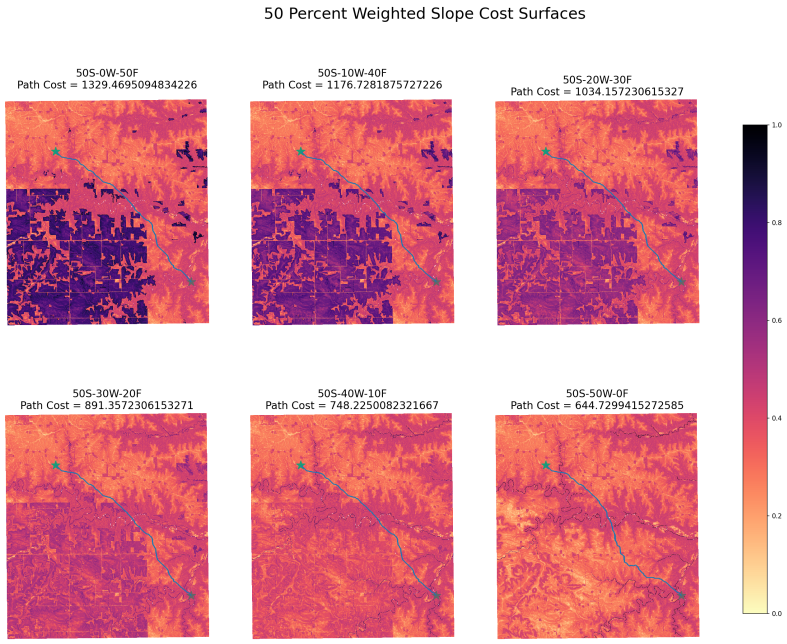

Figure 29: Cost Surfaces with 50% weight on slope cost.

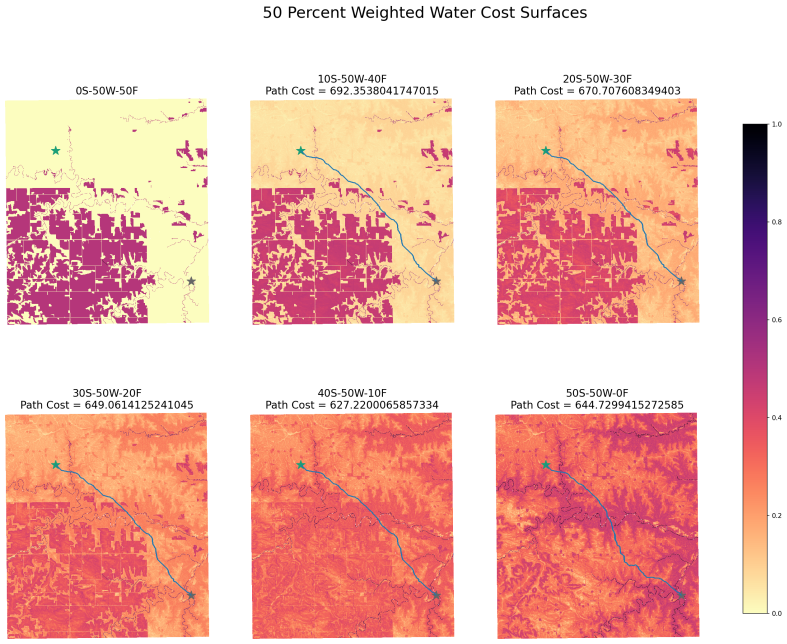

Figure 30: Cost Surfaces with 50% weight on water cost.

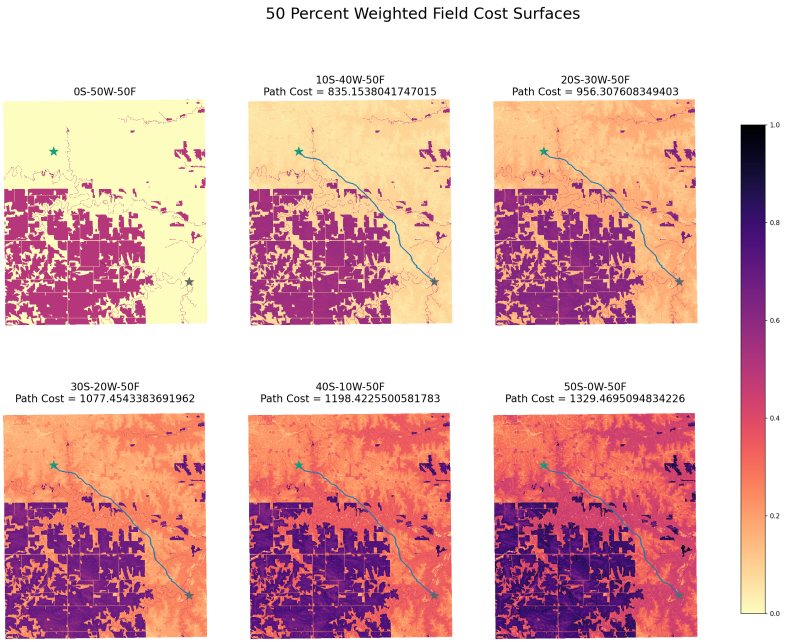

Figure 31: Cost Surfaces with 50% weight on field cost.

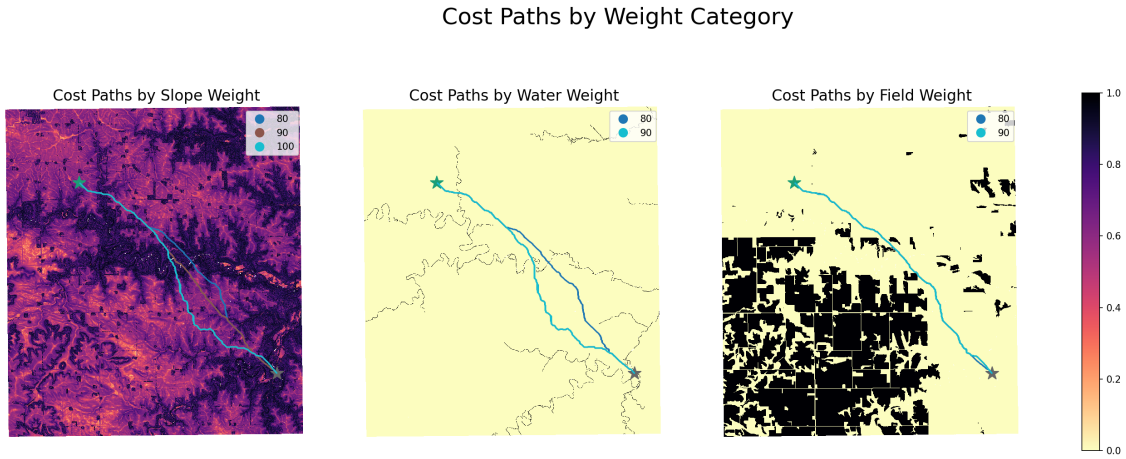

Figure 32: Cost paths for the most extremely weighted surfaces plotted over each cost layer.

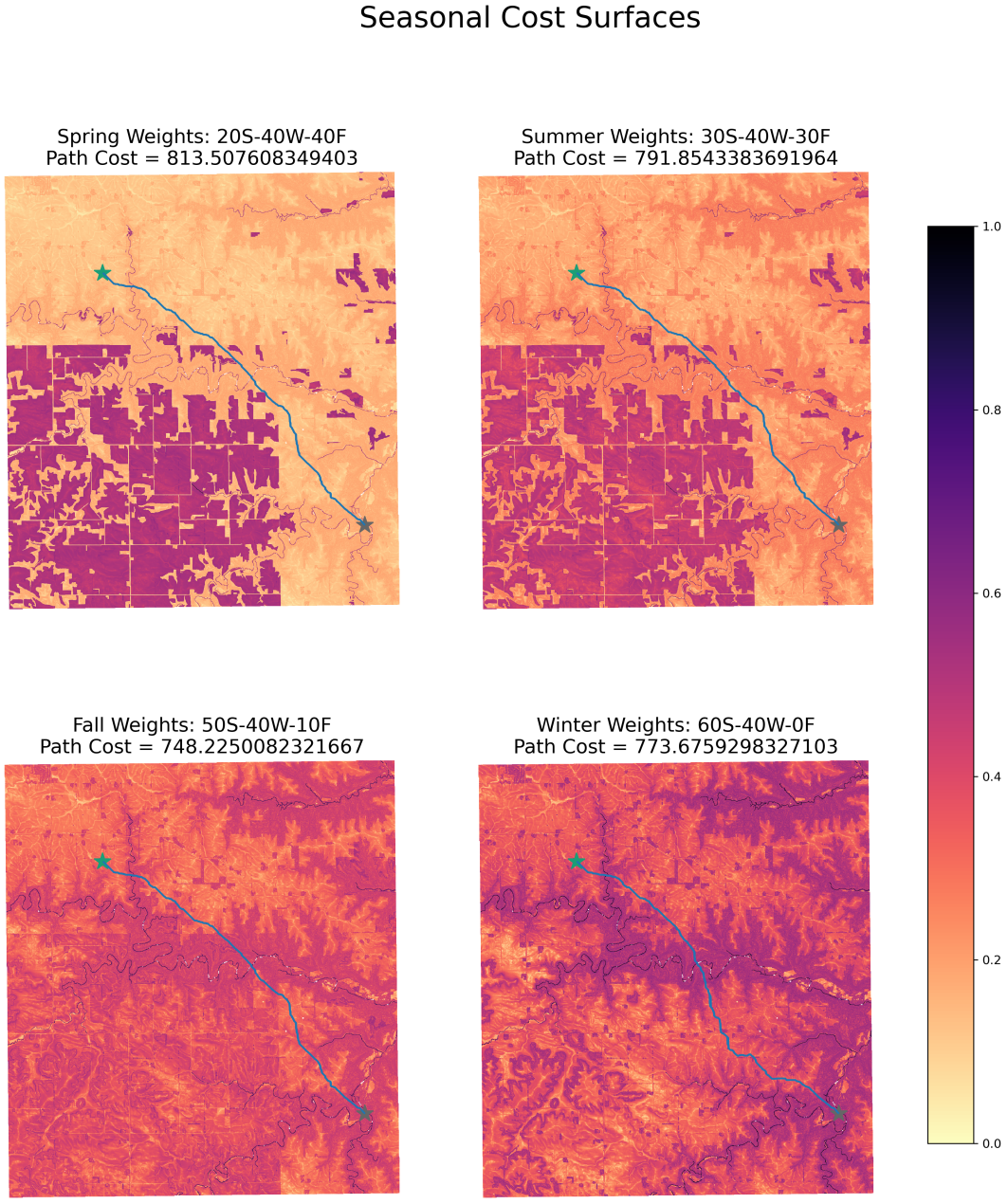

Figure 33: Proposed cost surfaces for each season.

Visually the results appear to be accurate.

5 Discussion and Conclusion

In this lab I took the next steps to improve the work done in Lab02. This involved improving the water cost layer and calculating least cost paths.

Since the water layer was full of holes in the last assignment, I decided to go ahead and use some official bridge and water data. This payed off quite well and created a pretty realistic water layer. The data that was employed definitely skipped over all the secret bridges that I'm sure Dory knows about, though.

All in all, there were 18 unique cost paths produced by the 66 cost surfaces and none of them varied far from the path of most gradual slope. I think this goes to show that perhaps the layers weren't weighted properly. For example, none of the cost paths utilized a bridge and the water layer only really altered the end of the trip (see figure 32, Cost Paths by Field Weight). The fields did affect the path significantly, however (Compare winter to other seasons in figure 33). Don't even get me started on cost!

Upon completing this lab, I am much more familiar with cost path analysis!

Self Score

Category	Description	Points Possible	Score
Structural Elements	All elements of a lab report are included (2 points each): Title, Notice: Dr. Bryan Runck, Author, Project Repository, Date, Abstract, Problem Statement, Input Data w/ tables, Methods w/ Data, Flow Diagrams, Results, Results Verification, Discussion and Conclusion, References in common format, Self-score	28	24
Clarity of Content	Each element above is executed at a professional level so that someone can understand the goal, data, methods, results, and their validity and implications in a 5 minute reading at a cursory-level, and in a 30 minute meeting at a deep level (12 points). There is a clear connection from data to results to discussion and conclusion (12 points).	24	22
Reproducibility	Results are completely reproducible by someone with basic GIS training. There is no ambiguity in data flow or rationale for data operations. Every step is documented and justified.	28	28
Verification	Results are correct in that they have been verified in comparison to some standard. The standard is clearly stated (10 points), the method of comparison is clearly stated (5 points), and the result of verification is clearly stated (5 points).	20	15
		100	89