Distance to sawmills code documentation

NFS Site Suitability Tool, Summer 2025  
Author: James Jin

data\_prep.py

Lines 12-24: setting up environment and reading in arguments

* Arguments include:
  + Workspace (File GDB)
  + Transportation dataset (feature dataset within File GDB)
  + Roads feature class (from OSM, using the roads\_data\_prep.py script)
  + NFS roads feature class
  + Sawmill feature class
  + Harvest site feature class
  + [optional] Boundary feature class
* Workspace and transportation dataset must be created beforehand, with the transportation dataset having the same spatial reference that all other files are to be projected to

Lines 27-64: projecting and clipping

* All feature classes are projected to NAD 1983 StatePlane Mississippi East
  + This can be changed within the script to be a different EPSG code
  + Each of the resulting projected feature classes are put into the File GDB workspace. It is recommended that the original data is stored in a separate directory/File GDB as the name of the feature class is not changed.
  + Either shapefiles or feature classes in a File GDB can be inputted
* If a boundary feature class is provided, it is also projected then used to clip the sawmills, NFS roads dataset, and harvest site dataset to be within the boundaries.
  + OSM roads data is not clipped as the boundaries for roads data is described in roads\_data\_prep.py

Lines 66-71: cleaning sawmill data

* All entries that have their status as closed or announced are remoed

Lines 73-205: merging roads and NFS roads feature classes, creating and building network dataset

* First, NFS roads data had duplicate roads removed. If more than 80% of the NFS road matches with a road in the OSM roads feature class, it is removed.
* The NFS roads data need to have a unique ID to identify the road.
  + This was done by generating 26 points along each NFS road
    - Done using the Generate Points Along Lines tool using the Percentage option at 4%, end points included
    - Can decrease percentage/increase number of points for more precise threshold
  + The Near tool was used on the generated points to determine if a point was close to a road (within 150 feet) (lines 67-71)
    - Used 0 or 1 in a IS\_NEAR field to indicate if a points is close to a road
  + Then it is determined if for a given NFS road if more than 80% of the points are marked with a 1, meaning the points are near an OSM road
    - The points are iterated through. Each original ID is put into a dictionary as the key, where the value is the sum of the IS\_NEAR field for all points corresponding that original ID.
    - Then the NFS roads are iterated through. If when accessing the dictionary using the ID of the NFS roads results in a value greater than 20 (>80%), then that line in the NFS roads feature class is marked as a duplicate
    - All roads in NFS\_roads not marked as a duplicate are exported into a “cleaned” roads feature class
  + The final roads feature class is then put together and made into a network dataset
    - The snapping process starts with generating end points at the end of each NFS road. Near is then used to determine which of the end points are near an OSM road. A point feature class is created from the near x and y values.
    - Before snapping the NFS roads to the near x and y values, NFS roads that do not connect to OSM roads are removed. This is done through a while loop.
      * In this while loop, NFS road end points with a negative NEAR\_DIST are selected and organized into a dictionary. In this dictionary, every end point’s ORIG\_FID is put in as the key with the value as the number of end points that aren’t near an OSM road.
      * A spatial join between the NFS end points and NFS roads is created so that every point that intersects with more than one NFS road is selected in a selected. For each of the selected points, if the ORIG\_FID of the selected point exists in the previously made dictionary, the value is reduced by 1.
      * For each NFS road ID in the dictionary that has a value of 2 (meaning both end points are neither touching OSM roads or another NFS road) are removed and a count for number of entries removed is incremented.
      * When the while loop repeats, the count is reset to 0 and the process repeats, stopping when the count is 0 at the loop start. This would mean that there are no more unconnected roads to remove.
    - The cleaned NFS roads are then snapped to the previously made points from the near x and y values.
    - The two road feature classes are merged together.
      * If the final roads feature class needs to be overridden and a network dataset has already been created, the network dataset is deleted before the merge
    - The Integrate and Feature To Line tool are also used on the final merged result to ensure connectivity.
    - A distance field is added to the roads feature class and the distance is calculated in miles
    - Using the roads feature class, a network dataset is created and built

Changes within this script

* Originally, a cost raster was made in this script to be used in Spatial Analyst tools, using a polyline to raster conversion with the distance as the value represented in the raster
  + This was scrapped as the Cost Path and Cost Distance tools proved to be rather inaccurate and inefficient
* The original plan also included splitting up the total distance calculation into 3 parts: harvest site to NFS road, NFS road to public road, then public road to sawmill. In the original version of this script, points were generated where the NFS roads would exit onto public roads
  + This fell through as merging the two road datasets proved to be simpler and it was falsely assumed that every harvest site would be connected to a NFS road.
* Sawmills used to be snapped to roads.
  + This was dropped as it was unnecessary, network analyst does this already
* Original ID for generated points used to be added manually
  + This was unnecessary, Generate Points Along Lines does this automatically as long as the NFS roads dataset comes with an ID
* It was attempted to remove OSM roads instead of removing NFS roads
  + Caused a much longer runtime due to a reversal of the process, where generating points was done on the much larger OSM road feature class
  + Still not clear if this was necessary, depends on if NFS roads fields prove valuable in determining path
  + May revisit if necessary
* Added harvest site clipping since pivoting to Activity\_TimberHarvest
* The threshold for if a NFS point counts as near was increased to 170 feet
* The sawmill dataset changed. The new forisk dataset contained closed and announced but yet to open sawmills. To ensure only open sawmills can be accessed, closed and announced sawmills were removed.
* There were issues with connectivity. To solve this, two changes were made:
  + Snapping was changed from simply snapping to either vertex or edge to manually creating snap points from Near x and y values to ensure better connectivity
  + The merged roads dataset had to be integrated and converted back into a line to ensure connectivity
* Originally there were issues with building inside the script. Upon trying again later, those issues didn’t present themselves, so building the network dataset is now done inside the script.
* The process of removing unconnected NFS roads was done because it was causing issues for some harvest sites that would path to the unconnected NFS roads, causing in no solution to be found.
  + The while loop had to be used to continuously remove roads as there was an instance of two roads unconnected to OSM roads but connected to each other. A solution to this problem could not be found that wouldn’t also impact other NFS roads that were valid. Removal in multiple iterations was implemented to solve this.

roads\_data\_prep.py

Lines 13-22: export\_to\_arcgis(edges, file, layer)

* Inputs:
  + Edges from a graph
  + GPKG file to export to
  + Layer name for feature class conversion
* Converts edges of a graph to a GPKG file
* Converts GPKG to a feature class in a File GDB
* Removes fields that have the Big Integer data type

Lines 24-57: reading in inputs and creating graph

* Inputs:
  + File GDB for roads
  + File GDB for nodes (not currently used)
  + Area of interest (e.g. “Mississippi, USA”)

- or -

* + North, South, East, and West values for bounding box
  + [optional] a specification of road type
* From the AOI or bounding box values, a graph is created using a custom filter
  + Road types are filtered for motorways, trunks, primary roads, secondary roads, tertiary roads, residential roads, and unclassified roads
    - This can be changed to fit specific needs
  + Optionally, if a specific road type is created, a graph using the single road type is also created
* The edges and nodes are extracted from the graph, though currently only edges are used

Lines 60-65: call export function to convert edges to roads feature class to be used in data\_prep.py

* Exports single type road graph edges as well if that option was used

Changes within this script:

* Custom filter originally included all road types
  + Was changed to remove road types unlikely to be accessible to logging trucks

slope\_raster.py

Lines 14-48: strip\_z\_and\_project(stream\_input, streams\_dir\_input, stream\_dataset\_output, spatial\_ref)

* Inputs:
  + Stream input feature class
  + Stream input directory
  + Stream feature dataset to output to
  + Spatial reference object
* Because streams data came with a z value, projecting to EPSG: 2899 was not possible. To remedy this, this function creates a new feature class with the same features. However, the features in the new feature class use the same x and y coordinates but leave out the z value. This new feature class is then projected to EPSG:2899
* This function may not be necessary depending on what the data is being projected to

Lines 50-79: stream\_setup(ws, str\_ds, str\_dir, spat\_ref, bd=None)

* Inputs:
  + Workspace
  + Stream dataset
  + Stream directory
  + Spatial reference object
  + [optional] boundary for clipping
* First each stream feature class in the stream directory is projected using the strip\_z\_and\_project() function (line 53-58)
  + List of feature classes is obtained using ListFeatureClasses() specifying for line feature classes
* Next the streams data is clipped and merged (depending on the input) (lines 61-74)
  + If boundary is provided, then each stream feature class is clipped by the boundary feature class. The bounded streams feature classes are then merged.
  + If no boundary is provided, then everything in the stream dataset is merged into one streams feature class
* A buffer of 100 feet is created around the merged streams data (lines 77-79)
  + The name of the streams buffer is returned

Lines 81-94: roadless\_area\_setup(ws, rl\_a, spat\_ref, bd=None)

* Inputs:
  + Workspace
  + Roadless area polygon
  + Spatial reference
  + [optional] boundary
* Projects and clips (if boundary is provided) the roadless area polygon feature class
* Returns the file name for the modified roadless area polygon feature class

Lines 96-108: create\_off\_limit\_areas(ws, merge\_list, roads)

* Inputs:
  + Workspace
  + List of feature classes to merge
  + Roads feature class
* Merges the list of off-limit feature classes
* Creates a 50 foot buffer around roads, then erases that buffer from the merged off-limit areas feature class
* Returns the name of the off-limit feature class

Lines 110-146: create\_slope\_raster(ws, elev\_data, ofa, bd, spat\_ref)

* Inputs:
  + Workspace
  + DEM
  + Off-limit areas feature class
  + Spatial reference object
  + [optional] boundary
* Projects the DEM raster to the designated spatial reference
* If there is a boundary, the DEM raster is clipped
  + The boundary feature class is buffered by a mile so that the algorithm can find roads outside of Bienville boundaries
  + The Clip Raster tool does not clip exactly as intended, as it uses a feature classes’ bounding box rather than the actual polygon. Instead, the boundary feature class is rasterized, the using raster algebra, a new raster is created where only the boundary raster and DEM raster exist
* The areas in the off-limits feature class area removed from the slope raster
  + First a slope raster is created from the DEM raster
  + Then the off-limits feature class is used to remove areas from the slope raster
    - This is done by rasterizing the off-limits feature class, then creating a mask the slope raster only exists where both the off-limits raster is Null and the slope raster exists. This mask is saved to the workspace

Lines 148-168: reading in inputs and using functions to create slope raster

* Inputs:
  + Workspace (File GDB)
  + Stream dataset (feature dataset within File GDB)
  + Stream directory (contains original stream feature classes)
  + Roadless area feature class
  + Road feature class
  + DEM raster
  + [optional] boundary
* First a list of items to be merge are created using the stream\_setup() and roadless\_area\_setup() functions. This list can be adjusted based on what inputs actually are needed to be added to the off-limit areas.
* Next the off-limit areas feature class is created using the create\_off\_limit\_areas() function
* Finally, the slope raster is created using create\_slope\_raster() function
* This script is set up like this so individual parts can be run separately and so that the actual input of roadless area polygons and streams data can be customized

Changes within the script:

* No major changes to the functionality of the script just yet, however there may be changes to how roadless area polygons are handled as this script is mainly handling streams data. It depends on what data is available.
* A buffer of the boundary was added to the process for creating the slope raster so that when the least cost path is being calculated, roads outside of Bienville can be considered.

BEFORE DISTANCE CALCULATIONS CAN BE RUN, DO THIS  
Oneway functionality must be manually implemented in ArcGIS Pro. To do so, follow these steps:

* Open the streets\_nd network dataset properties in the catalog
* Create a new travel mode called "Driving Distance"
  + Under costs, ensure length is used for impedance
* Under the costs tab at the top, ensure the distance field is used for the evaluators
* Under the restrictions tab, create a new restriction called "Oneway"
  + usage Type: prohibited (-1)
  + under evaluators, for the Along source, set the type to field script and use this value and code block
    - Value=evaluator(!oneway!, !reversed!)
    - Code Block:  
      def evaluator(oneway, reversed):  
       return oneway == 1 and reversed == 1
  + under evaluators, for the Against source, set the type to field script and use this value and code block
    - Value=evaluator(!oneway!, !reversed!)
    - Code Block:  
       def evaluator(oneway, reversed):  
       return oneway == 1 and reversed == 0
* Go back to the travel mode tab and for driving distance, make sure the Oneway restriction is checked
* Build one more time before calculating

Distance calculator

Lines 11-109: calculate\_distance(harvest\_site, roads, network\_dataset, sawmills, slope, off\_limit\_areas, output\_path, sm\_type=None)

* Inputs:
  + Harvest site
  + Roads feature class
  + Roads network dataset
  + Sawmill(s)
  + Slope raster
  + Off-limit areas
  + Output path
  + [Optional] sawmill type
* First, the distance from harvest site to nearest road is calculated based on slope using a least cost path algorithm
  + First the centroid of the harvest site needs to be calculated to act as the starting point
    - The Erase tool is used to remove areas of the site within the off-limit areas
    - The centroid is then calculated using the Feature To Point tool with the inside option checked to force a point inside the harvest site
  + The roads feature class is rasterized to be used as the destination
  + The least cost path is calculated using the defined function calculate\_least\_cost\_path() using the centroid, rasterized road, and slope raster (the function will be explained in more detail below)
  + The result is a path from the centroid to the road. However, the path does not quite extend all the way to the road feature class since it was rasterized. For the rest of the path, Near() was used to find the closest point on the road feature class to the end point of the path. In the unlikely case there are multiple points near the road (within 60 feet), the closest point is used. The x and y values from Near() are used to create a point that serves as the starting point for network analysis.
* Next the path from the previously determined point to the sawmill is calculated using Network Analyst tools.
  + One option is there is a single sawmill destination provided, which then uses the calculate\_road\_distance\_nd() function.
  + The other option is if multiple sawmill destinations are provided, in which case the function calculate\_closest\_road\_distance\_nd() is called, which takes in multiple sawmill destinations and finds the closest one.
    - If a sawmill type is provided, then the sawmill data will be filtered for that type.
    - If no sawmills of that type exist in the data, then an error will be raised
    - Alternatively, this option does not need to be used. Data can be filtered out beforehand if desired.
  + For both options, Euclidean distance is calculated using the function euclidean\_distance\_near().
    - Before this function can be called, if the multiple sawmill option was used, then after the route is found, the sawmill destination must be made into a path. End points are generated which are then used to filter sawmills by proximity to the end point of the route. The sawmill that is found by this process is used as point 2 for Euclidean distance calculation.
  + If no sawmill is found, an error will be raised.
* Combining the two paths and calculating total distance
  + The least cost path is snapped to the network path
  + The two paths are then merged into one feature class
  + Visualization of the two paths connected: A map of a road

    AI-generated content may be incorrect.
  + The resulting feature class is then used in the function calculate\_distance\_for\_shp(output\_path), which returns the distance of a polyline feature class
  + Both the total road distance and Euclidean distance are returned

Lines 112-129 calculate\_least\_cost\_path(starting\_point, dest, cost\_raster, output\_path)

* Input
  + Starting point (harvest site centroid)
  + Destination (road raster)
  + Cost raster (slope)
  + Output path
* The least cost path is found using Spatial Analyst tools Cost Distance and Cost Path. This path goes from the harvest site centroid, navigates through the slope raster, avoiding off-limit areas, and finds the nearest road. The resulting path is then converted to a polyline.

Lines 130-168: calculate\_road\_distance\_nd(starting\_point, network\_dataset, sawmill, output\_path)

* Inputs:
  + Starting point (on the roads feature class)
  + Network dataset of roads
  + Sawmill destination (singular)
  + Output path
* The road path is computed by make a Route layer and adding the starting point and sawmill destinations as stops. Then the Solve() method is run to find the route between the two. This route is then saved into the output path.
* This function will attempt to set the one-way road restrictions

Lines 170-213: calculate\_closest\_road\_distance\_nd(starting\_point, network\_dataset, sawmills, output\_path)

* Inputs:
  + Starting point (on the roads feature class)
  + Network dataset of roads
  + Sawmill destination (multiple)
  + Output path
* Instead of a route layer, a Closest Facility Analysis layer was made. This is done so that multiple sawmills can be added as facilities and Solve() will choose the closest one. Otherwise, this function works like the previous function.
* This function work for both single or multiple sawmill destinations but tends to be less efficient than calculate\_road\_distance\_nd(). When a specific sawmill destination is desired, it is recommended to use the other function instead.
* This function will attempt to set the one-way road restrictions

Lines 215-229: calculate\_distance\_for\_fc(fc\_path)

* Input:
  + Feature class path
* Adds a distance field to a polyline feature class. Uses CalculateGeometryAttributes() to calculate distance in miles.
* The distances of each line are summed up and returned

Lines 231-242: euclidean\_distance\_near(point\_1, point\_2)

* Input
  + First point (singular)
  + Second point (singular)
* Calculates the Euclidean (straight-line) distance between two points using the Near() method
* Returns the distance

Changes within the script:

* Originally, the primary distance calculation was done using the Spatial Analyst tools with the rasterized roads as the cost raster.
  + This was dropped as it was too inaccurate
  + The code was repurposed into least cost path analysis using slope for the distance from harvest site centroid to nearest road
  + The road distance was then calculated using Network Analyst
* There were multiple iterations of the path from harvest site centroid to nearest road
  + First, this path was estimated using a straight-line path from centroid to nearest road.
    - This was too much of an oversimplification of the path
  + Another option was to use the Cost Path tool but to use the road distance feature class as the destination
    - This had some unintended behaviors
    - The path did not always find the nearest road, sometimes crossing over one road to get to a different further road
  + The rasterized road proved to be the best destination to be used in the Cost Path tool. However, the resulting path did not connect fully to the roads feature class. The first attempt to remedy this was to snap the path to the roads feature class.
    - This did not always produce accurate results. Snapping to vertex and edge were both tried but neither worked universally.
    - Eventually, the final solution was to snap to the end of the calculated network path instead of the roads raster, which was able to produce suitable results
  + Originally, distance was calculated using calculate\_distance\_for\_fc() for each path produced by the smaller functions. However, this was changed to only run calculate\_distance\_for\_fc() after the merge into the final path feature class as the total distance could not be calculated before the snap.
  + Euclidean distance using Harversine formula was considered but was not as accurate as using Near. It also was not so much faster that it was worth using.
* When calculating the centroid, the original method was to erase parts of the harvest site that overlapped with off-limit areas, then find the polygon with the largest area, and use that polygon’s centroid as the centroid for the entire harvest site
  + This was initially done manually. However, using the “inside” option in Feature To Point does roughly the same thing, so for the sake of simplifying code, this option was used instead.
* Because the forisk sawmill dataset contained much less precise coordinates, the end resulting placement of sawmills are not completely accurate to what they actually are. To ensure that sawmills can be found by Network Analyst, search tolerance was increased to 2000 feet.
* For calculating road distance to closest sawmill, an enormously large number for the cutoff was added to the arguments of the MakeClosestFacilityAnalysisLayer() method to ensure that any sawmill can be found, no matter distance.
* The point on the road that represented the nearest point from the resulting least cost path was made into a feature class for testing purposes
* Due to the inaccuracy of the forisk sawmill dataset, when searching for the final stop in the route when multiple sawmills exist in the sawmill input dataset must be increased to 2000 feet. To ensure unwanted sawmills aren’t also found in this process, the end point of the route is generated and used as the center of the search area.
* Occasionally there is an error when calculating distance with calculate\_distance\_for\_fc() where the smallest line has no length value, which causes a TypeError. If this error is caught, the smallest line distance will be omitted since it is too small to make a significant difference.