Road crossing cost algorithm

Algorithm overview and implementation documentation

Vlk Jan

Robotic inspection of gas pipes project
with the Vision for robotics and autonomous systems group

FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Department of Cybernetics
Czech Technical University in Prague
Czech Republic
October 6, 2022

Abstract

It is quite common for autonomous mobile robots to encounter a road in their path that they need to cross. In this paper we focus on developing a simple cost function for any road that splits the road into smaller segments and determines the suitability of each segment for road crossing.

We have limited ourselves to road curvature, classification and elevation profile as parameters that determine the crossing cost. In addition we have limited ourselves to the road itself, we did not consider the effects of buildings or other envinronmental structures near roads on the line of sight. We also did not take into account pedestrian crossings as this cost function was part of a bigger project where pedestrian crossings were dealt with separately.

1 Introduction

During the preparation phase of this project we figured out that our robots would have to cross roads during their runtime. In order to safely traverse to the other side we have to detect incoming vehicles in advance. This puts constraints on the places where we can cross safely as many places do not provide sufficient line of sight. The line of sight of our robots may be impaired by curves, buildings, trees or other terrain irregularities. It is also not advisable to cross near road junctions as the traffic situation can be confusing and unreadable.

We focused on cost calculation based on the roads curvature and the terrain irregularities of the roads itself (the elevation profile of said road). As last part we added the cost of road classification where lower class roads are more suitable for crossing as vehicles tend to roam them more slowly and generaly have less lines our robot would have to cross.

2 Algorithm overview

This algorithm was designed to evaluate and set costs for individual road segments based on their suitability for crossing. It consists of multiple steps. First one is identifying roads in osm data[1] of the target area. Second is calculating intermediate cost functions based on different requirements. One evaluates curves and distances from them as well as distances from road junctions, other evaluates terrain irregularities and their effect on visibility, and the last evaluates the suitability for crossing based on road classification. Last step combines the previous three cost functions into final crossing cost, which is the output of our algorithm.

Algorithm 1 An overview of our algorithm.

```
Input: $map$ ... OSM map data of area of interest
Input: $elev\_data$ ... Elevation data of area of interest
  $road\_network \leftarrow$ process $map$
  $class\_costs \leftarrow$ set classification costs
  $curve\_costs \leftarrow$ set curvature costs
  $elev\_costs \leftarrow$ set elevation costs
  $costs \leftarrow$ combine $class\_costs$, $curve\_costs$ and $elev\_costs$
Output: $costs$
```

Geometrical insight

In order to properly construct our algorithm and more importantly to correctly set constants that manage the run of our algorithm we need to understand the situation we are trying to solve. The most important limitation of our robots is the distance in which thay can recognize incoming vehicle. This limitation is therefore based on the lidar and camera properties. When we examined the footage from both devices we found out that the view distance of lidars we use is about 40 m

[1]Open street map data downloaded with Overpass API.

and the visibility of our cameras is about 80 m. With visibility we mean that at the distance stated we are able to detect vehicles and other obstacles.

We assume that the minimal width of a road is 3 meters for one way roads and 4 meters for two way roads. With this information we may assume that it takes approximately 10 seconds for our robots to cross the road. Having precise measurement of road width would help us in path planning and it would give us one more parameter to base our calculations on. While open street maps allow street widths to be specified, most roads do not come with this information, and therefore we have to plan without it.

The last information we consider is the height in which the sensors are located. For our robots the camera and lidar are approximately at waist height meaning the visibility distance may be reduced by smaller terrain irregularities than it would take to restrict the visibility of a human. With this in mind we created the values for two parameters for TPI classification S_limit and L_limit.

2.1 Road identification

Identifying road segments is relatively easy as all road ways have a “highway” tag and generaly speaking are ways[2]. We therefore select all ways that this applies to and prepare them for easier manipulation. From these roads we later create road network.

We also detect all intersecting points of individual road objects. From these intersection we later select all points that mark junctions, junctions are intersection points where there are at least three roads connecting. To these points we add points representing T-shaped junctions, which we detect as road end node connecting with a non end node of another road.

Algorithm 2 Overview of map processing.

```
Input: $map$ ... OSM map data of area of interest
  $roads \leftarrow$ find all roads in $map$
  $intersections \leftarrow$ detect intersecting points in $roads$
  $junctions \leftarrow$ select junctions from $intersections$
  $roads \leftarrow$ combine disconnected roads
  $road\_network \leftarrow$ create road network from $roads$
Output: $road\_network$
```

2.2 Road curvature

The idea behind this part of the cost function is that near junctions and curves we cannot see incoming vehicles very clearly. Making road segments near these features less suitable for crossing than lets say segments of a long straight road.

We already have the positions of all junctions in our road network from previous step.

The evaluation of the road curvature is based on radii of circumcircles of triangles made from points on road in which the road changes direction. Figure 1 shows why the circumcircle radius is being used to determine road curvature. Equation (1) shows how to calculate the circumcircle radius.

$$r = \frac{(a \cdot b \cdot c)}{\sqrt{(a+b+c)\cdot(a+b-c)\cdot(a+c-b)\cdot(b+c-a)}}, \tag{1}$$

where a, b, c are the lengths of respective triangle sides.

As we know the precise coordinates of all three vertices, we are able to calculate all neccessary values to later evaluate the radius.

We have 6 levels of “curvature”. The highest level is reserved for segments near junctions, other segments curvature levels are determined from their circumcircle radius. The curvature value is then set by the segments length multiplied by the segments curvature level weight.

[2]One of the three elements used by osm $\rightarrow$ nodes, ways and relations.

The final value of segments cost is then calculated in following manner. We take a segment and look in both directions of travel from the segment. We go in one of the directions and sum the values of segments curvature we encounter until we get to an exploratory range limit or we reach the end of the road, then we do the same in the other direction. In the end we divide the sum by the length of the explored road, floor the value and set it as the final cost of curvature for this segment.

Algorithm 3 Overview of curvature cost calculation.

```
Input: $road\_network$ ... Representation of all roads in target area
  $radii \leftarrow$ calculate circumcircle radius for all road segments in $road\_network$
  $segments \leftarrow$ rank segments based on their $radii$
  $curve\_costs \leftarrow$ calculate curvature cost based on $segments$
Output: $curve\_cost$
```

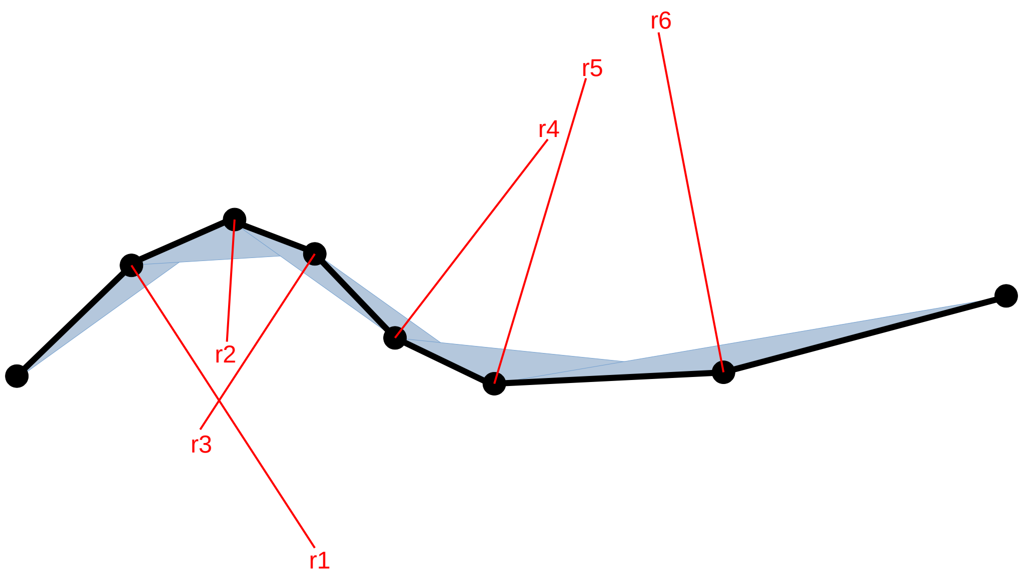

Figure 1: Visualization of circumcircles radii in road curvature evaluation.

2.3 Road elevation

The idea behind this part is based on reduced visibility near terrain irregularities.

We take the road network that we have prepared in first step and divide each road into finer equidistant segments. We than assign heights to the dividing points, the height assignment takes five nearest points from the elevation data[3], and calculates the height in following manner

$$e = \sum_{i=1}^{5} e_i \cdot \frac{\sum_{j=1}^{5} d_j}{d_i}, \tag{2}$$

where e_i is elevation of the i-closest point and d_i is the distance of these points from our desired point.

After we have heights assigned to road points we classify each point with TPI landform classification. We use a modified version of the classification that uses two intervals (smaller and larger one). The main difference is that instead of two-dimensional interval we only use one-dimensional

[3]We use data from the Czech Land Survey Office – ZABAGED® 5G.

interval, and the limiting constants are slightly altered. The equation (3) shows how we calculate the TPI value we later use to assign TPI class.

$$TPI = e_0 - \frac{\sum_{i=1}^{N} e_i}{N}, \tag{3}$$

where e_0 is elevation of the point of evaluation, e_i is elevation of points within boudaries, N is the number of total points in the evaluation.

For all points we have two values, one for smaller (SN) and one for larger (LN) interval. Based on these two values and limiting constants we can classify each point as one of ten different classes. Five of them are shown in figure 2 to help with understanding of the meaning of individual classes. The classes are:

- Canyons – SN: TPI $\leq$ −S_limit, LN: TPI $\leq$ −L_limit
- Midslope drainages – SN: TPI $\leq$ −S_limit, LN: −L_limit $<$ TPI $<$ L_limit
- Upland drainages – SN: TPI $\leq$ −S_limit, LN: TPI $\geq$ L_limit
- U-shaped valleys – SN: −S_limit $<$ TPI $<$ S_limit, LN: TPI $\leq$ −L_limit
- Plains – SN: −S_limit $<$ TPI $<$ S_limit, LN: −L_limit $<$ TPI $<$ L_limit, slope $\leq 5°$
- Open slopes – SN: −S_limit $<$ TPI $<$ S_limit, LN: −L_limit $<$ TPI $<$ L_limit, slope $> 5°$
- Upper slopes – SN: −S_limit $<$ TPI $<$ S_limit, LN: TPI $\geq$ L_limit
- Local ridges – SN: TPI $\geq$ S_limit, LN: TPI $\leq$ −L_limit
- Midslope ridges – SN: TPI $\geq$ S_limit, LN: −L_limit $<$ TPI $<$ L_limit
- Mountain tops – SN: TPI $\geq$ S_limit, LN: TPI $\geq$ L_limit

We have a cost associated with each of these classes, and the final cost of the segments is evaluated in the same way as final evaluation for road curvature with only one change, here we do not count the cost of the segment we are evaluating.

Algorithm 4 Overview of elevation cost calculation.

```
Input: $road\_network$ ... Representation of all roads in target area
Input: $elev\_data$ ... Elevation data of target area
  $roads\_elev \leftarrow$ calculate elevation profile for all roads
  $TPI\_classification \leftarrow$ classify road segments with TPI classification
  $elev\_cost \leftarrow$ calculate costs base on $TPI\_classification$
Output: $elev\_cost$
```

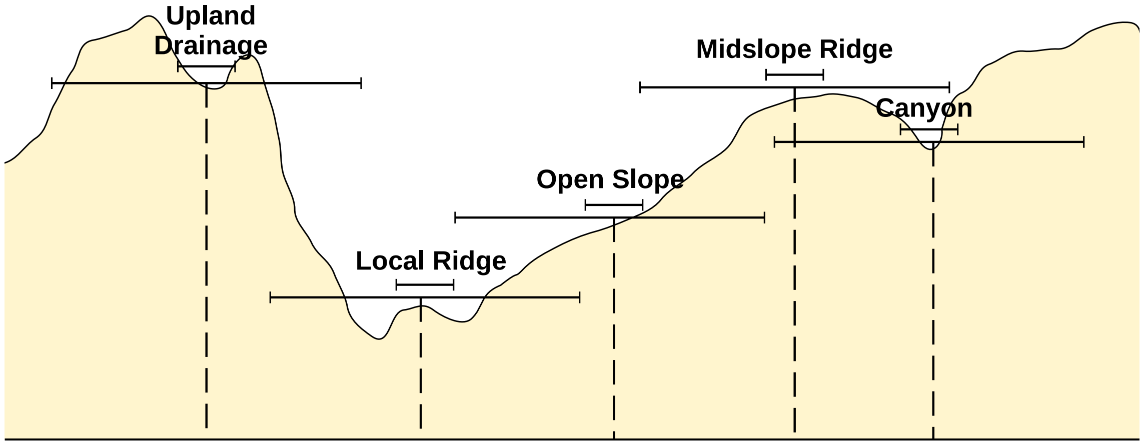

Figure 2: Visualization of a few TPI landform classes.

2.4 Road classification

The idea behind this part of cost algorithm is that lower tier roads are more suitable for crossing than higer tier ones. Generally speaking on lower tier roads cars are driving more slowly and less frequently. Lower tier roads also usually have less lines than higer tier ones. The osm provides multitude of different highway classifications some of them we identify as footways and do not consider them a road that needs to be crossed carefuly.
The following road classification are the ones we evaluate, roads on the same line have the same cost associated with them:

- Primary, primary link
- Secondary, secondary link
- Tertiary, tertiary link, busway
- Residential, living street, road, service
- Unclassified

2.5 Algorithm constants

We can easily modify the function of the algorithm by tweaking the following constants, these constants may be foung in a file road_crossing_consts.py. More about individual constants in chapter 3.4.

- Weights for the 6 levels of road curvature
- Maximal radius for each level of road curvature
- Cost for individual road classifications
- Number of levels in individual parts of road crossing cost algorithm (curvature, elevation, classification, final)
- Weights of intermediate cost functions relative to final cost
- Number of points in batches for elevation assignment (impacts speed)
- Interval and limiting constants for TPI landform classification
- Costs for individual types of TPI landform classes

3 Algorithm implementation

The whole algorithm is divided into several files for easier manipulation.
The files in which the algorithm is divided into are:

- road_detection.py
- road_curvature.py
- road_elevation.py
- road_crossing_consts.py

3.1 road_detection.py

In this file there are methods used for creation of road network, finding intersections and junctions and overall preparation of osm data for further work.

3.1.1 get_roads(data)

This method returns list of all roads we consider crossable, meaning they have one of the following higway tags: primary, primary_link, secondary, secondary_link, tertiary, tertiary_link, unclassified, residential, service, busway, road, living_street.
The input data is the osm data query we get from Overpass API of the target area.
The return value is a list of tuples, each tuple represents one road in the osm data. First value of the tuple is list of tuples, these tuples represent nodes the road is made of, first value of the tuple is the node's id, second value is the node's lattitude and third value is the node's longitude. Second value of the road tuple is the road's classification.

3.1.2 gps_to_utm(data, withID)

This method returns a numpy array of gps lattitude and longitude coords transformed into UTM format.
The input data is a list of tuples with coords in gps lat and lon to be transformed into UTM.
The input withID is a bool value that specifies if the tuples in input data list have an ID in the first position. The default value is true.
The return value is a numpy array where the lat and lon values have been transformed into UTM format.

3.1.3 create_line_for_road(road, withID)

This method creates shapely.geometry.LineString object for road nodes.
The input road is a list of tuples with coords in UTM format.
The input withID is a bool value that specifies if the tuples in input data list have an ID in the first position. The default value is true.
The return value is a LineString object that has been created from the road nodes coordinates.

3.1.4 create_road_network(roads, inUTM, withID)

This method creates shapely.geometry.MultiLineString object for all roads in the osm data of target area. We firstly create a LineString object for individual roads and then combine them into one MultiLineString object. This object is the road network structure we use in further parts of the algorithm.
The input roads is a list of lists of tuples. Each list inside this list represents one road for which we create a LineString object.
The input inUTM is a bool value that specifies if the coords in roads are in UTM format. The default value is true.
The input withID is a bool value that specifies if the tuples in input data have an ID in the first

position, this value is not used if inUTM is false. The default value is true.
The return value is a MultiLineString representing the road network in our area of interest.

3.1.5 find_intersections(road_network)

This methods finds all intersection points in our road network. This is achieved with inbuild shapely.geometry methods. We only deal with Point and MultiPoint objects, as there should not be any other kind of intersection (such as LineString). These intersecting points are not necessarily junctions as roads in osm data are often split into multiple ways and we currently treat all LineString objects as individual roads.
The input road_network is a MultiLineString object that we created in previous method.
The return value is a list with Point objects that represent every intersecting point in our road network MultiLineString.

3.1.6 find_junctions(intersections, road_network)

This method chooses junctions from intersections. All intersection Point objects that are present in three and more LineString objects are junctions. We find the rest of junctions if we take an intersection and look if it is only present in end points of LineString objects, if it is not then we know that it is also a junctions.
The input intersections is a list with Point objects that we marked as intersections in previous method.
The road_network is a MultiLineString object representing our all roads in the area of interest.
The return value is a list of Point objects representing junctions in our road network.

3.1.7 combine_road(junctions, intersections, road_network)

This method takes points that are in intersections but are not present in junctions, meaning that these points are places where a road was split in two. We than connect the two corresponding LineString objects into one. In this method we also set auxiliary points to equidistantly split the road in such a manner to not have two points more than 20 meters apart.
The input junctions is a list of Point objects representing junctions in road network.
The input intersections is a list of Point object representing all intersecting points of LineString objects in our road network.
The input road_network is a MultiLineString object representing our road network.
The return value is a MultiLineString object, it is a modified input road_network where all roads start and end in a junction.

3.1.8 road_class_price(roads)

This method takes roads and based on their classification returns the cost of crossing.
The input roads is a list of tuples, first value of tuple represents the road nodes, second is the roads classification.
The return value is a list of tuples, each tuple represents one road. First value in the tuple is a LineString object generated from the roads nodes and second value is the roads cost based on its classification.

3.1.9 visualize_curves(segments, crossings, grid)

This method is used as a way to see which road parts are marked as which level of curvature. It displays the colored road network in a matplotlib window. This method is only used as a debugging tool.
The input segments in a list of lists, each of these lists has numpy arrays in them that represent each equidistant road segment. One value in the dictionary is a curvature level, which is the value we want to display.
The input crossings is a list of points representing crossings, it does not serve a purpose now as we

deal with crossings in a different way.
The input grid is a bool value that specifies if we should show grid in final matplotlib image. The default value is true.
This method does not have a return value.

3.1.10 visualize_curvature_rank(ranked_segments, crossings, grid)

This method is used as a way to visualize the final road crossing cost based on curvature. There are 15 prepared colors, so any number of levels below or equal to 15 will work, if you intend to show more than 15 levels this function must be modified. This method is only used as a debugging tool.
The input ranked_segments is a list of lists, where each list represents one curvature rank level.
The input crossings is a list of points representing crossings, it does not serve a purpose now as we deal with crossings in a different way.
The input grid is a bool value that specifies if we should show grid in final matplotlib image. The default value is true.
This method does not have a return value.

3.2 road_curvature.py

This file contains methods that calculate individual road segments curvature. And the crossing cost based on the curvature.

3.2.1 circum_circle_radius(A, B, C)

This method calculates the circumcircle radius for triangle defined by the points A, B, C.
The inputs A, B and C are tuples representing points that define the triangle we want to calculate the circumcircle radius for. In our usecase these points are the road nodes.
The return value is a floating point variable representing the circumcircle radius of the triangle defined by points A, B and C.

3.2.2 get_average_radius(road_network)

This method takes the road network of the target area and sets radius for each road segment. It does not set the average, rather it takes the minimal radius of the two nodes connected to the segment.
The input road_network is a MultiLineString object representing all roads in the target area.
The return value is a list of numpy arrays, each array represents one road. The array has a custom dtype of curve_type = dict(names=names, formats=formats), where names = ['radius', 'length', 'coords', 'curvature_level', 'curvature'] and formats = ['f8', 'f8', '(2,2)f8', 'i1', 'f8'].

3.2.3 rank_segment_curves(segments, junctions)

This method assigns curvature levels depending if the road segments contains junction node or based on the circumcircle radius.
The input segments is a list of numpy arrays which represent individual roads in the road network. The arrays have a custom dtype of curve_type, defined in chapter 3.2.2.
The input junctions is a list of all junction nodes in the road network.
This method does not have a return value.

3.2.4 road_cost_for_curve(segments, exploration_limit)

This method takes the curvatures of road segments and combines them into final curvature crossing cost. How we compute the final value is explained in chapter 2.2.
The input segments is a list of numpy arrays which represent individual roads in the road network. The arrays have a custom dtype of curve_type, defined in chapter 3.2.2.

The input exploration_limit is an integer variable defining the maximal distance we will explore in one direction.
The return value is a list of tuples, where the first value is a LineString object representing the road segment and second value is the final curvature crossing cost.

3.3 road_elevation.py

This file contains methods that classify elevation profile and calculate crossing cost based on the classification.

3.3.1 generate_waypoints(road, waypoint_density, circular)

This method is used to generate equidistant points in a road based on prefered point density.
The input road is a LineString object representing road in which we want to generate points.
The input waypoint_density is a floating point varibale determining the maximal distance between two points.
The input circular is a bool value that specifies if the road is closed (meaning first and last points are the same). The default value is false.
The return value is a list of tuples, where each tuple represents a point along the road.

3.3.2 get_road_network_elevation(road_network, elev_data_files)

This method takes the road network, generates equidistant waypoints and sets elevation for each of these generated points.
The input road_network is a MultiLineString object representing all roads in the target area.
The input elev_data_files is a list of strings, which represent the paths and file names of the files containing elevation data of the target area.
The return value is a list of lists of tuples, each list represents a road and each tuple contains x, y and z coordinate values.

3.3.3 classify_TPI(elev_data)

This method classifies the elevation profile with the TPI landform classification.
The input elev_data is a list of lists of tuples, each list represents road and tuples consist of x, y and z coordinates of road nodes.
The return value is a list of lists of tuples, each list represents a road and tuples are constructed from tuple of x and y coordinates and TPI classification.

3.3.4 road_cost_for_height(network_classification, exploration_limit)

This method calculates final road crossing costs for all road segments based on the elevation profile, namely on the TPI classification.
The input network_classification is a list of lists of tuples, each list represents a road and tuples are constucted from tuple of x and y coordinates and TPI classification.
The input exploration_limit is an integer variable defining the maximal distance we will explore in one direction.
The return value is a list of tuples, first value in the tuple is a LineString object of the particular road segment, second value is the final elevation crossing cost.

3.4 road_crossing_consts.py

This file contains constants that modify basic functionality of our algorithm. We will show the names of all constants with their base values.

3.4.1 Road detection

In this cathegory we only have constants that specify costs for different road classifications.
PRIMARY_ROAD_COST = 5
SECONDARY_ROAD_COST = 4
TERTIARY_ROAD_COST = 3
UNCLASSIFIED_ROAD_COST = 1
RESIDETNATL_ROAD_COST = 2
SERVICE_ROAD_COST = 2
BUSWAY_ROAD_COST = 3
ROAD_ROAD_COST = 2

3.4.2 Road curvature

In this cathegory we have constants determining the maximal curvature for individual levels of curvature level and weights for each one of the six curvature levels.
LEVEL_1_MAX_RADIUS = 220
LEVEL_2_MAX_RADIUS = 150
LEVEL_3_MAX_RADIUS = 90
LEVEL_4_MAX_RADIUS = 40

LEVEL_0_WEIGHT = 0.75
LEVEL_1_WEIGHT = 3
LEVEL_2_WEIGHT = 6
LEVEL_3_WEIGHT = 10
LEVEL_4_WEIGHT = 20
LEVEL_5_WEIGHT = 30

3.4.3 Road elevation

In this cathegory we have constants determining the amount of points in one elevation determining cycle, TPI evaluation intervals and change values, weights for every TPI class and elevation data files.
NODES_IN_BATCH = 10000

SMALL_NEIGH = 3
LARGE_NEIGH = 7
SMALL_CHANGE = 0.7
LARGE_CHANGE = 1.2

CANYONS = 8
MIDSLOPE_DRAIN = 5
UPLAND_DRAIN = 6
U_VALLEY = 2
PLAINS = 1
OPEN_SLOPES = 3
UPPER_SLOPES = 6
LOCAL_RIDGE = 8
MIDSLOPE_RIDGE = 10
MOUNTAIN_TOP = 15

ELEV_DATA_FILES = None

3.4.4 Miscellaneous

In this cathegory we have all other constants that did not fit in any other cathegory. Such as number of levels in individual cost steps and weights of each individual cost step.
ROAD_CURVATURE_RANKS = 15
ROAD_CLASS_RANKS = 5
ROAD_ELEVATION_RANKS = 9
ROAD_CROSSINGS_RANKS = 15

CURVATURE_WEIGHT = 70
CLASS_WEIGHT = 5
ELEVATION_WEIGHT = 25