TIME CRITICAL TARGET: ROAD NETWORK CODE

K. C. THOMPSON
SEPTEMBER 27, 2016

Time Critical Target (TCT) is a program that seeks to provide effective and efficient Intelligence, Surveillance, and Reconnaissance (ISR) modeling and simulation solutions in an effort to derive analyses and statistics to help support decision making in a given scenario. Many scenarios represent historical, realistic, and idealized accounts of military events and their defensive and/or offensive strategies. TCT solutions integrate platform, communication, sensor, resource manager, and tracker components to simulate these scenarios over a defined time period. In many cases there is a target (or targets) of interest to be pursued. These targets are desired to be tracked to not only gauge intent, but to also prevent catastrophic damage/fatalities. Tracking must be performed accurately, as time is "critical" in capturing the target.

While tracking a target is a very difficult task, it becomes more manageable when the target is constrained to a road network. Road networks can dictate how a target moves, the speed it travels, and the maneuvers that it makes. Using road network geometry, we may be able to quickly recover a lost track on a target and better predict where the target might be provided that it has moved out of a sensors field-of-view. This paper discusses the road network code developed to aid TCT efforts to utilize road network analysis to enhance track and search algorithms. The code is a refinement of techniques discussed in [1]. The code is developed in C++, and features KML parsing, KML-to-graph, roadNetwork-to-GOG, and road network analysis components.

If interested there is a C++code available to process Open Street Map (.osm) data (author Brian Powell), and a MATLAB prototype to demonstrate "Targets Moving on a Road Network".

1. Code Structure

The code has 5 primary components, using supporting packages and structures defined in header files. The working directory includes each of the files defined below, and contains a build directory in which the program should be executed using `CMake` and `Make` tools.

(1) `get_kml_list.cpp` - gathers a list of all kml files in the working directory.
(2) `kml_to_graph.cpp` - transforms a kml file in the list generated from `get_kml_list.cpp` into a `RoadNetwork` struct that contains both a vector and BOOST graph representation of the kml file.
   (a) `check_road_nodes.cpp` - iterates through the RoadNetwork vector and adds nodes whenever the distance between nodes on the same road vector exceeds a certain distance.
(3) `roadNetwork_to_GOG.cpp` - transforms each `RoadNetwork` vector element generated by `kml_to_graph` into a general overlay graphics (GOG) file. GOG files are specifically supported in SIMDIS[1], and are used to show more information on a map.

[1]SIMDIS is a tool integrated with the TCT's Closed Loop Collaborative ISR Simulation Testbed (CLCSIM) to create two and three-dimensional interactive graphical and video display of live and post processed simulation, test, and operational data.

(4) `graph_analyzer.cpp` - a list of functions that operate on the graph and/or vector elements of a `RoadNetwork` struct to compute information about the associated road network.

(5) `main.cpp` - iterates over the list generated by `get_kml_list.cpp` while calling `kml_to_graph.cpp` and `graph_to_GOG.cpp`.

(6) `RoadNetwork.hpp` - defines all packages, structures, components, and functions used to declare elements related to a `RoadNetwork` struct.

(7) `CMakeLists.txt` - the input to the CMake build system for building the RoadNetwork package defined above. Because Digital Terrain Elevation Data (DTED) is used, `CMakeLists.txt` links to CLCSIM's PEMT[2] geo-data libraries.

1.1 `get_kml_list.cpp`

In general, `get_kml_list.cpp` is a convenient way to process unrelated KML files in one execution. The program looks for all *.kml* files in the working directory and generates a list to be processed by `kml_to_graph.cpp` which gets passed in `main.cpp`. In addition, this script was designed for the convenience of translating multiple KML files to GOG files (visual representation of KML). Often times the KML file representing the road network extracted from a geographical location is huge. Larger files result in longer processing times. To prevent this, "prior" to execution the KML can be broken up into smaller KML files and processed piece by piece. Hence instead of waiting several minutes for one GOG file, smaller GOG files will be outputted at shorter time increments.

Although convenient, when invoking `get_kml_list.cpp` be careful to only store the KMLs of interest in the working directory. Otherwise, processing the KML list could take up a lot of time provided numerous files exist in your working directory. Alternatively, this section of the code can be commented out, and made to process a single KML.

1.2 `kml_to_graph.cpp`

The program takes in a KML file name and generates and returns the `RoadNetwork` struct associated with the KML file in the working directory. The KML file is processed by finding lines that contain "<coordinate>...</coordinate>", and parsing data between those tags with BOOST tokenizers. The data is parsed into the `RoadNetwork` struct *road_map* vector, which is later used to create the `RoadNetwork` BOOST graph *GRAPH* element. The *road_map* is a vector R of vectors r_i of tuples r_{ij}. Each v_i represents a road within the road map. Each v_{ij} represents a coordinate or position (lat, lon, alt) on the road. Often times the KML files read in will contain coordinates "clamped" to the ground i.e., with zero altitude. However, for adequate 3D rendering in SIMDIS correct altitude values need to be computed. If not, vehicles, sensors, and objects associated will be placed below the ground which could further prevent detections. Hence, if necessary `kml_to_graph.cpp` will invoke DTED libraries and compute and update the altitude value within the tuple.

After the *road_map* vector is created, its nodes are then assessed by `check_road_nodes.cpp`. This program was created to check the sparsity of the KML file. KML files acquired from our collaborators may represent a broad set of way points; hence, the distance between nodes can vary largely. Having a more uniform distance across nodes increases road network coverage for initial placement of sensors and/or pseudo-tracks. In our analysis, node distance will also impact target movement. Consider the case of targets having to stop at each node. For

[2]PEMT is a CLCSIM component that provides implementations of many ISR processing and control algorithms, as well as many commonly useful utility APIs.[2]

similar targets (traveling the same speed), uniform grids can bring all targets to a stop at once which may be a great startint point for initial analysis efforts.

In [4], it is noted that the minimum spacing between standard guide signs should not be less than 800 feet on rural freeways and 600 feet on urban freeways; for non-freeways, the minimum spacing is 350 feet in rural areas and 200 feet in urban areas. Being careful not to overload the road network, `check_road_nodes.cpp` defaults the spacing for new placement of nodes to $0.24383km$ ($800ft$). This value can be changed within `kml_to_graph.cpp` when `check_road_nodes.cpp` is called.

TAfter the *road_ map* has been refined the *GRAPH* element can be constructed. First *GRAPH* needs a node identifier for each unique node. Note that all coordinate values have been truncated to six decimal places. The sixth decimal place is worth up to $0.11m$: you can use this for laying out structures in detail, for designing landscapes, and building roads [3]. Using a BOOST `set`, unique nodes are identified and placed into two BOOST maps:

(1) *coordtoNode* - map with `key`=coordinate and `value`=node ID, and
(2) *nodetoCoord* - map with `key`=node ID and `value`=coordinate.

There may be an easier way to do this, but here two maps are used to provide the capability to extract uniquely associated elements forward and backward. Some *GRAPH* components such as *V_ PositionMap* (vertex position map) and *EdgePair_ vec* (edge pair vector) need access to the unique coordinate given the node ID (or vertex ID); some need access to the unique node ID given the coordinate e.g, *GRAPH Points*. The node ID is a numerical value $1 \ldots N$. The node ID is also used as the label for the nodes in the *GRAPH*. `get_kml_list.cpp` builds a bidirectional *GRAPH* by defining its vertices, vertex positions, edges, edge lengths, and edge slopes.

1.3 `roadNetwork_to_GOG.cpp`

Recall that a GOG file is a general overlay graphics file that SIMDIS supports for bringing more realism to your visualization. In general, GOG files are not necessary as SIMDIS will render KMLs. To load a KML in SIMDIS, on the toolbar select Map, GOG, and then Load or Tool. Within Tool you can change the color and line width of the KML along with a few other things. However if the KML does not "adequately" reflect the road network (two points defining a long stretch of road), importation of these files directly will lead to poor representation of the road network within SIMDIS. In this case, points can be added to the *road_ map* to fill in gaps and/or holes within the road network.

If the coordinates within the KML have been altered, a GOG file or any other compatible graphics file is needed to render the updated KML. Node insertions generated from the KML will be reflected in the `RoadNetwork` elements, and passed to `roadNetwork_to_GOG.cpp`. `roadNetwork_to_GOG.cpp` takes in the KML file name and the associated *road_ map*, and produces the GOG file. The structure of this code was developed by Brian Powell, and modified to handle newly defined `RoadNetwork` elements. Within this code, road network linewidth and color can be modified; the default is `linewith`=3 and `linecolor`=yellow. Note that these variables can also be changed within SIMDIS after loading the GOG file.

1.4 `graph_analyzer.cpp`

References

[1] Powel, B. P. 2016. Road Network Graph Construction from KML and OSM Formatted Data. *Johns Hopkins Applied Physics Laboratory* Distribution. KVW-6-16U-001.

[2] DeSena, John T. 1997. Closed-Loop Collaborative ISR Simulation Test Bed (CLCSIM). *Johns Hopkins Applied Physics Laboratory* Github. http://desenjt2/data/share/README.ad.html.

[3] Measuring Accuracy of Latitude and Longitude. *Geographic Information Systems*. http://gis.stackexchange.com/questions/8650/measuring-accuracy-of-latitude-and-longitude.

[4] Traffic and Safety Design Division. September 2016. *TRAFFIC SIGN DESIGN, PLACEMENT, AND APPLICATION GUIDELINES.* Michigan Department of Transportation. http://mdotcf.state.mi.us/public/tands/Details_Web/mdot_signing_design_placement_application_guidelines.pdf.