Range Queries for Symbolic Trajectories on Road Networks

Nathan Aguirre, Ben Meline
Work performed under Dr. Goce Trajcevski

Summary of Approach

In this project, we report our method of determining if a path exists between semantic trajectory points, such that the path passes through some defined region at a certain time. First, we needed a way to generate the road network. Using data from OpenStreetMap we were able to generate a graph representing downtown Chicago. Next, we added points of interest to this road network, which could potentially be traversed. Then we needed a way to match trajectory points to the road network. We were able to do this by finding the closest node in the road network to a particular trajectory point. Given two nodes we then needed to develop a path searching algorithm. We created an algorithm that searches for possible paths between two points, given a polygon and a time interval. Search continues until a path is found that meets the constraints, or all possible paths are exhausted. Lastly, we created a method to read trajectories from a text file, and the test if the trajectories are possible for a polygon and time interval.

Introduction and Solution

Semantic trajectories provide a format for describing the movement of some object in an informationally-minimized way. In our project, these semantic trajectories provide a start point, an endpoint, and points along the way where the object stops. Given this type of information, it is possible to extract possible routes from the start to the end without specifically following a time-parameterized version of a trajectory. This allows for much lower required storage space. [1]

The difficulty with using semantic trajectories, of course, comes from the fact that between two points, there may be many possible paths. Therefore, they can be used to narrow down or cluster similar explicitly parameterized trajectories. For this project, we examine semantic trajectories in the context of roadmaps. We are interested in determining which semantic trajectories meet some criteria. Specifically, we seek to determine which semantic trajectories could possibly result in the associated object (a vehicle/person) being inside some pre-defined polygonal region within some time-interval. [1,2]

Semantic trajectories, for our purposes, fall under two commands: “moves” and “stops.” A “move” command indicates that an object move between some source point *S* to some target point *T*. *S* and *T* are given in terms of latitude and longitude. A “stop” command indicates that an object at some point *P* did not move from *P* for some duration t_{wait}. Therefore, a semantic trajectory can be described as a series of move and stop commands. Knowing this full trajectory, we first seek to determine the routes between the first point in the trajectory and the last point in the trajectory, passing through all explicitly-stated in-between points, waiting at any points indicated by a “stop” command. [2]

Given just this minimal representation of a semantic trajectory, it is clear we must constrain the problem. For even a moderately large graph, this path-finding problem can have extremely high time complexity. Because we are attempting to determine which trajectories have a non-zero probability of being inside a polygonal region within some interval, we therefore must test many possible spatial pathings for a given semantic trajectory. This leads to the first constraint on our problem: we will only consider simple paths (paths without repeated node visits) when searching paths. This significantly reduces the possible spatial paths to consider.

While typical semantic trajectories present more specific time points for moves, we have made a modification to our inputs, due to limited data and the remaining resources at our disposal.

Using OpenStreetMap, a platform that allows users to submit street information such as latitude, longitude, road type, etc., we were able to generate a road network surrounding downtown Chicago. [3] However, while the platform allows speed limits to be tagged, there are very few cases in which this tag is used in our network. Therefore, we prescribed a mean speed for all streets based on the type of road it is (‘highway_tag’). As such, these grossly-approximated speeds do not necessarily correlate to the simulated semantic trajectories we have been provided with. To modify this, we simplify our inputs and make an assumption: Given a semantic trajectory with source point *S*, target point *T*, and maximum time t_{max}, does the semantic trajectory have a point in which the object is located within a polygon *K* within some time interval (t_1, t_2) while being able to reach *T* at or below time t_{max}.

While this analysis is fairly straightforward for a trajectory with a single move command, it is complicated by stops. For our analysis, we will consider only a limited list of possible stop points described by a name, as well as its latitude and its longitude. We will refer to these as “points of interest” or POIs. Therefore, for each stop command in a semantic trajectory, we inject the respective points into our graph. From this, we determine the edge closest to the POI coordinates, then the point on the edge that is closest to the POI. If the closest point on the edge is one of the end nodes, we simply assign attributes poi_name and wait_time to the node and keep track of this node. [4] If the node is nearest to some point on the edge, then we project the POI to that nearest point, create a new node ‘poi_name’ with attribute wait_time, remove the original edge and generate new edges between the new node and each of the end nodes. These new edges retain the original edge’s direction, type, and speed, but with recalculated values in accordance with the altered distance.

Once this new graph has been generated, we perform a depth-first search on the graph, looking for paths which contain nodes that are inside the given polygonal region and are within the lower and upper bounds of the time window.

In order to drastically reduce the runtime of our program, we perform several calculations on the graph prior to path finding in order to eliminate impossibilities, preventing a hanging algorithm in as many cases as possible. In order to do this, for every waypoint along a semantic trajectory, we calculate three items: the shortest time to the polygon $t_{\text{to poly}}$, the shortest time to the next point along the trajectory $t_{\text{to target}}$, and the shortest time to the target with a path that goes through the polygon $t_{through\ poly}$. Knowing a time window (t_1, t_2) and a time limit t_{limit} we can look at these calculated calculated values. Refer to Fig. 1 below for a schematic of these.

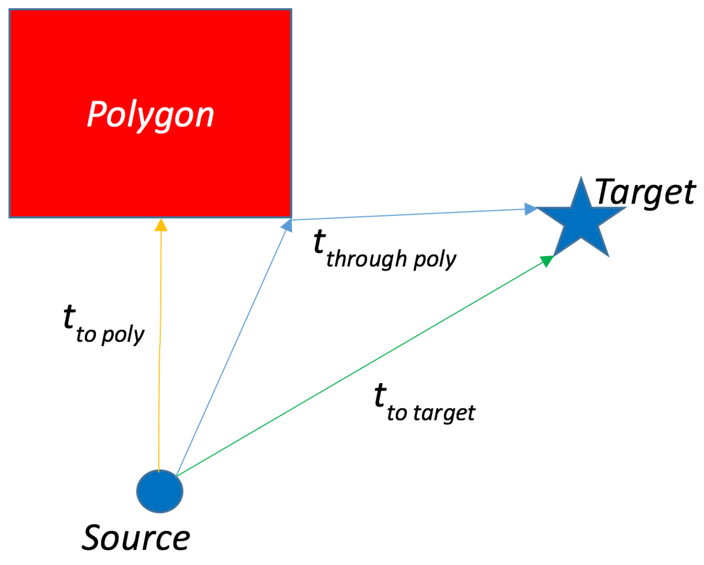

Fig. 1: Some smallest distance metrics used to determine invalid paths. For example, if $t_{through\ poly}$ is greater than the time limit, then this trajectory would not be possible. These metrics are also used for each leg of the trajectory, e.g. if $t_{to\ poly}$ is greater than the upper time bound on our window, then we would not be able to find a path using that leg.

Code Description

Three text files, a query file, a POI file, and a trajectory file, are needed as an input to our Python 2 implementation. The query file contains the polygon vertices, the upper and lower time bounds, and the time limit. The POI files contains the name of each point of interest along with the latitude, longitude, and wait time. The trajectory file is derived from Tian Zhang's format, with each line containing an index for the full trajectory, a 'move' or 'stop' indicator, as well as the latitudes and longitudes for the source and target nodes of that leg (time points contained therein are not used). [4] A graph is generated from the collected OpenStreetMap data using the Python package NetworkX. [5] POIs are injected as described above.

For each node in the trajectory, we calculate shortest paths relative to the source, the target, and the polygon using a modification of Dijkstra's algorithm. Using these shortest path calculations in tandem with the given time constraint allows our algorithm to stop searching paths when it is clear the trajectory would be impossible.

To find a path between two nodes we used an iterative depth-first search of the graph. A stack is used to keep track of the nodes that need to be visited. The children of the current node being tested are pushed onto the stack and then at each iteration a node is popped off the stack. The runtime up to the current node is tracked as well as the nodes visited up to the current node. To continue following the current path the runtime needs to remain within the time range given by the query. Time that is spent at a point of interest is added to the current time as well as time spent traversing an edge. If the path is inside the polygon within the time interval then the find path function returns back the path up to this point. If a path is over the time interval or the total time limit then it is popped of the stack, so that a different path can be attempted.

Results

Below are a couple visualized examples of the program. Nodes within polygons are marked in red. Single points of other colors mark nodes along the trajectory, while paths are indicated by a colored sequence of connected nodes. If no path for the respective color is shown, then no path was possible with the input semantic trajectory and the time constraints. Note that only a portion of the paths are displayed; this is because we have previously calculated that there exists a shortest path from each previous node or to each subsequent node that stays within time constraints.

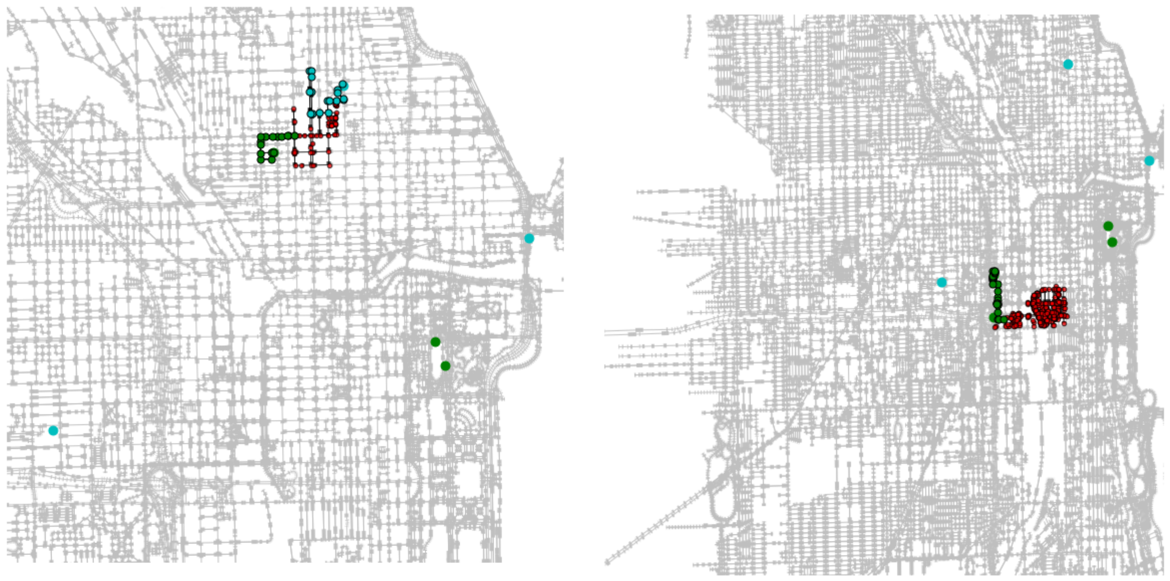

Fig. 2: Two examples of valid path results. At left, two trajectories in green and cyan. Both start very close to the the polygon P (red) and are able to find a path that leads to a node inside the polygon within time range (0,0.02) and are able to reach remaining nodes in their trajectory within a very large time limit. At right, only one trajectory (green) is able to find a path to the polygon within time constraints.

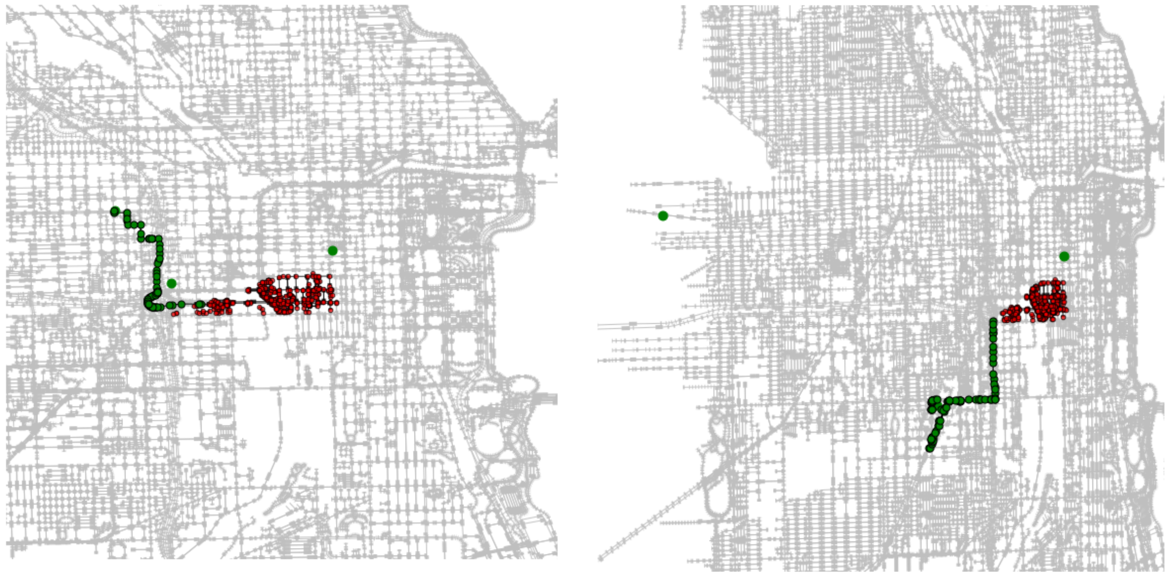

Fig. 3: Two versions of the same parameters, with only the first and second points on the semantic trajectory changed. In the left image, there exists a path from the start node to the polygon within time constraints. In the right image, the start node has been moved to the extreme left and the second node lower and to the left. While there does exist a path from the start node to the polygon within the time constraints, the full path that would include both the second node and final node is unable to be reached within the time limits. A path from the start node to the second node to the polygon to the final node, however, is applicable, and we see that portion of the path returned.

Future Work

The solution we developed is able to determine if a trajectory is possible, given some constraints. There are multiple ways that our solution could be improved and extended.

Currently if an edge passes through the polygon, but there is no node inside the polygon the path finding algorithm does not detect that the trajectory was inside the polygon. This could be an issue if the polygon is small and the road network does not have a node inside the polygon. To solve this a node could be added where the edge intersects the polygon. Then it would be possible to detect if a path intersects the polygon.

Our solution returns a binary value for a particular trajectory, whether it is possible or not. An extension to this could be to return a probability for a trajectory of the path being inside the polygon within the time range. This would require searching all possible paths and determining which ones met the requirements. The runtime for this algorithm would be considerably longer because it is not just returning as soon as a successful path is found, which is how the solution currently works.

Lastly, the runtime could be improved by using some other data structure to perform the query. Searching all possible paths in a graph takes a long time and there isn't a way to determine if a path is headed in the right direction. Using a kd-tree could possibly make the solution more efficient. A kd-tree could perform a search using location and time information.

Conclusion

In this project we created a system to perform range queries on semantic trajectories using time and location. Trajectories, which include points of interest, are matched to a road network. The edges of the road network include speed, distance and time information. If we are given the vertices of a polygon we can determine which nodes in the network are inside the polygon using ray casting. We implemented an algorithm that walks the graph starting at a particular node and determines if there is a path that passes through the polygon in some time in the time interval. Our solution tries multiple paths until it runs out of possible paths. The runtime of the solution could be improved by using a different data structure, such as a kd-tree. An improved solution could also return a probability of a trajectory passing meeting the constraints, instead of just returning whether it’s possible or not.

References

1. Renato Fileto, Vania Bogorny, Cleto May, Douglas Klein. Semantic enrichment and analysis of movement data: probably it is just starting! 11-18. SIGSPATIAL Special, 7(1), 2015.

2. Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady L. Andrienko, Natalia V. Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, José Antônio Fernandes de Macêdo, Nikos Pelekis, Yannis Theodoridis, Zhixian Yan. Semantic trajectories modeling and analysis. ACM Comput. Surv. 45(4): 42 (2013)

3. OpenStreetMap.org

4. Tian Zhang. Generate Trajectories with POIs inserted.

5. NetworkX v1.11