Big Data Computation of Taxi Movement in New York City

Joya A. Deri, Franz Franchetti, and José M.F. Moura
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213 USA
Email: {jderi,franzf,moura}@ece.cmu.edu

***Abstract*—We seek to extract and explore statistics that characterize New York City traffic flows based on 700 million taxi trips in the 2010-2013 New York City taxi data. This paper presents a two-part solution for intensive computation: space and time design considerations for estimating taxi trajectories with Dijkstra's algorithm, and job parallelization and scheduling with HTCondor. Our contribution is to present a solution that reduces execution time from 3,000 days to less than a day with detailed analysis of the necessary design decisions.**

***Index Terms*—New York City open data, design, performance, high-throughput computing, HTCondor**

I. Introduction

Understanding traffic flow in cities has significant ramifications, from optimizing daily commutes to facilitating evacuations in the case of emergencies and natural disasters [1], [2]. In particular, the tempo and pattern of traffic flow need to be known in order to implement policies that can handle traffic congestion or unforeseen events. With the advent of smart city initiatives such as New York City's Open Data project [3], traffic data such as camera data [4] and taxi data [5] provide new ways of examining the movement of city traffic, opening new research in areas such as big urban data visualization and analytics [1], [2], [6], [7].

This paper focuses on designing software tools to extract features from 2010–2013 New York City taxi data, which consists of 700 million taxi trips for 13,000 medallions (taxi cabs) with data fields including pick-up and drop-off GPS coordinates and timestamps [5]. Since the raw data does not include taxi trajectories, computation of the taxi paths is essential to extract features that reflect taxi movement at each node and edge of a road network. Other studies based on tracking taxi GPS data over a road network include [8], [9], [10], [11]. Some of these studies use GPS traces to discover anomalous trajectories; others use GPS data to model and predict traffic and its evolution. In [12], betweenness centrality measures that incorporate traffic levels inferred from GPS data are computed over a transportation network in order to determine optimal locations of traffic monitoring units.

This work was partially supported by NSF grant CCF-1513936.

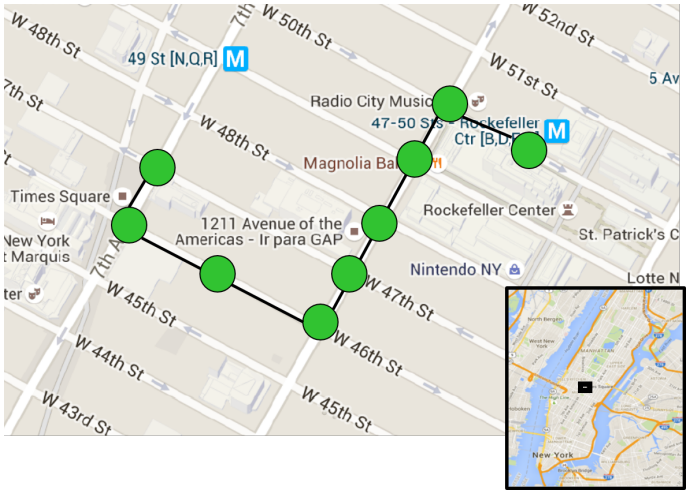


Figure 1. Dijkstra shortest path of a single taxi trip from Times Square to the Rockefeller Center. Our problem requires computing 700 million such paths and extracting relevant statistics with a low-latency, workable solution. (Map data: Google [14])

Our objective is to leverage the entirety of the data to extract taxi-based statistics over the Manhattan grid, such as taxi concentrations at road intersections. Such intensive data processing, which is necessary to learn and extract knowledge from large data sets or historical repositories such as the taxi data, requires solutions that are efficient in both memory utilization and execution time so that the large scale nature of the available data can be fully exploited [13]. In particular, we design a low-latency, memory-efficient solution to compute statistics to describe New York City taxi movement.

Dynamic representation. A dynamic representation of the taxi trips is needed in order to represent taxi movement in New York City. Since the available data provides only static (start and end) information for each trip, taxi trajectories along the road network are estimated. These trips are approximated by shortest paths computed with Dijkstra's algorithm, as shown in Figure 1. These paths are used to extract statistics of interest, such as the average number of trips that pass through a given location at a given time of day, the average number of passengers of these trips, and the average tip paid. Figure 2 illustrates such statistics.

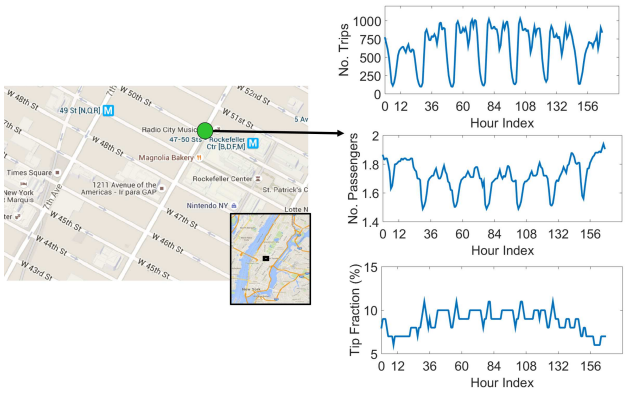


Figure 2. Examples of four-year average statistics computed at an intersection close to Rockefeller Center: average number of trips (top right), average number of passengers (middle right), and average tip fraction (bottom right). The hour index corresponds to an hour of the week with index 0 corresponding to Sunday 12am, index 12 to Sunday 12pm, index 24 to Monday 12pm, etc.

Data discovery. The richness of the New York City taxi data permits us to extract statistics that provide an in-depth characterization of taxi trips. Since the statistic of interest may vary with the research question, we desire an efficient solution for feature extraction that enables interactive data discovery.

Our goal is to design a feature extraction scheme for the taxi data so that the time to solution is reasonable. On one hand, real-time solutions are desirable, although they may not be necessary for data discovery on a historical repository. On the other hand, waiting weeks or months for statistics is not practical. For our research application, a computation that takes less than a day is acceptable and can be reduced with more resources (e.g., more powerful cluster machines or more cluster machines). Our solution design takes this constraint into account.

For extracting features from the New York City taxi data, the shortest path computation becomes a bottleneck. To illustrate, a Python implementation on a 64-bit, 16-core, 16 GB RAM machine takes one hour to compute Dijkstra paths for 10,000 taxi trips, which scales to 3,000 days to compute all 700 million trips. The computation time can be reduced to a few weeks by parallelizing with high-throughput computing resources such as HTCondor [15], [16]. However, this turnaround is not ideal for a data discovery problem such as the one we aim to solve. Instead, we would like to have the time to solution to be less than a day as discussed above. In this paper, we present such a solution in the C programming language and illustrate parallelization considerations on a memory-constrained computer cluster of 32 16-core/16 GB and 8-core/8 GB machines.

Contributions. This paper makes the following contributions:

- We demonstrate a solution that reduces computation from 3,000 days to less than a day.
- We recast the problem as a two-pass problem.
- We present our solution for a space-efficient, portable C implementation.
- Our solution is parallelizable for HTCondor.

Related software tools and approaches are discussed in Section II. The problem formulation and engineering constraints are described in detail in Section III. Section IV provides the implementation with experimental results in Section V. Results that demonstrate fine-grained, localized taxi trip descriptors as a function of spatial coordinates and time are shown in Section VI.

II. Background

This section provides background on computing platforms and path planning algorithms related to efficient computation of Dijkstra shortest paths.

Related frameworks. Scientific computing languages such as Python [17] and MATLAB [18] as well as new computing languages such as Julia [19] provide platforms for modeling complex data structures such as graphs and for computing statistics. One of our primary constraints at runtime is memory, which requires an implementation that eschews data structures in favor of simple array-based representations. For this reason, we implement our solution in the C programming language.

In addition, a low-latency, high-throughput platform is desired in order to run many jobs in parallel. High-throughput computing platforms for Big Data include MapReduce [20], Hadoop [21], and Spark [22]. The GraphX library in Spark can be used for parallel and distributed graph processing [23]. The individual jobs are designed to handle the required graph processing in order to use HTCondor, an open-source software tool that provides a high throughput computing environment on a cluster of machines [15]. HTCondor works well when each job is designed to have a low memory footprint.

Path planning algorithms. Dijkstra's algorithm [24] is related to shortest path algorithms such as breadth-first search, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm [25], [26]. Breadth-first search can be used to find a shortest path on an unweighted, directed graph. The Bellman-Ford algorithm finds shortest paths from a single source on graphs that are both weighted and directed, but it computes distances to multiple destination nodes. The Floyd-Warshall algorithm computes all-pairs shortest paths, takes $O(|V|^3)$ time, and can be optimized as in [26]; it can be modified to find a single path but is usually used for dense graphs.

This paper focuses on Dijkstra's algorithm [24], [25], which performs single-source single-destination path-finding on a sparse, directed network with distances as (non-negative) edge weights. Dijkstra's algorithm can run in $O(|E| \log |V|)$ time for sparse, strongly connected networks; more details are provided in Section III-B. Variations

on Dijkstra's algorithm include the A* algorithm, which partially computes a tree of paths to guide the search to the destination node [27], [28], [29]. In addition, methods such as pre-computing distance oracles [29], creating a hierarchical representation of the road network with contraction hierarchies [30], [31], or predicting subnetworks that contain the desired path by pre-computing cluster distances [27] have been shown to improve performance with a space trade-off. Furthermore, Google Maps [14] can be queried for trajectories that account for congestion and other parameters, although these queries would reflect current instead of historical (2010-2013) traffic patterns.

We select Dijkstra's algorithm [24], [25] to estimate taxi trajectories because the benefits of storing pre-computed paths to accelerate computation were outweighed by the utility of implementing an algorithm with a low memory footprint for the purpose of parallelizing on the available cluster with HTCondor. While Dijkstra's algorithm may not reflect a true taxi trajectory, it provides an approximation that demonstrates the challenges of intensive data processing. Methods for improving the Dijkstra path computation to reflect true trajectories are discussed in Section VI. Related evaluations of Dijkstra's algorithm include [32], which provides a probabilistic analysis to compare priority queue implementations, and [33], which compares serial and parallel implementations of Dijkstra's algorithm.

This paper focuses on the design considerations for implementing Dijkstra's algorithm in a high-throughput environment that requires a low memory footprint. The necessary design decisions for the problem formulation are provided in the next section.

III. Problem Formulation

This section presents the problem of computing statistics for New York City taxi data over a road network. Section III-A discusses memory constraints and necessary design decisions to make the problem memory-efficient. The expected computation time and design decisions to reduce latency are discussed in Section III-B. Section III-C highlights the high-throughput nature of the problem.

Overall objective. Our goal is to extract features that characterize taxi movement through New York City from four years (2010-2013) of New York City taxi data [5]. However, since the path of each taxi trip is unknown, an additional processing step to estimate taxi trajectories is required before extracting statistics of interest. For example, if a taxi picks up passengers at Times Square and drops them off at the Rockefeller Center, the statistics of interest would capture not only trip data at the landmarks, but also at the intermediate locations as depicted in Figure 1.

Estimating tax trajectories requires overlaying the taxi data on the New York City road network. The taxi data and network are described next.

NYC taxi data. The 2010-2013 taxi data consists of 700 million trips for 13,000 medallions [5]. Each trip has about 20 descriptors including pick-up and drop-off timestamps, latitude, and longitude, as well as the passenger count, trip duration, trip distance, fare, tip, and tax paid. The data is available as 16.7 GB of compressed CSV files.

Road network. The road network $G = (V, E)$ consists of a set V of $|V| = 79,234$ nodes and a set E of $|E| = 223,966$ edges that can be represented as a $|V| \times |V|$ adjacency matrix A. The nodes in V represent intersections and points along a road based on geo-data from [34]. Each edge $(v_i, v_j) \in E$, $v_i, v_j \in V$, corresponds to a road segment on which traffic may flow from geo-location v_i to geo-location v_j as determined by Google Maps [14]. An edge of length $d_{ij} > 0$ is represented by a nonzero entry in the adjacency matrix so that $[A]_{ij} = d_{ij}$. A zero entry corresponds to the absence of an edge; i.e., no edges of length zero are present in the network.

The network G is *directed* since the allowed traffic directions along a road segment may be asymmetric. In addition, G is *strongly connected* since a path (trajectory) exists from any geo-location to any other geo-location.

Initially, computing statistics along each taxi trajectory appears to be a single-pass problem. However, as this section shows, performance can be improved by separating the trajectory estimation and statistics computation. The rest of the section discusses the reasons for such a workflow.

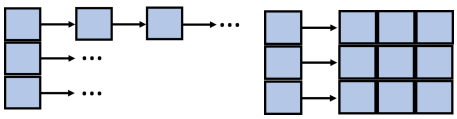
A. Memory-Efficient Design

The design choices are discussed for the backbone workflow, which consists of computing a taxi trajectory through the road network and then computing statistics from the path. A memory-efficient design is first demonstrated.

Road network. Methods such as creating a representation of the road network with contraction hierarchies or pre-computing cluster distances to predict subnetworks that contain the shortest path may be implemented so that the entire network is not in memory for each computation [27], [30]. On the other hand, for a machine cluster that does not provide a shared memory space, parallelizing shortest path computations requires the entire road network to be known at runtime to compute shortest paths, or that hierarchies or subnetworks are pre-computed and loaded at runtime. For the purpose of this paper, we focus on the case when the entire network is needed at runtime and discuss the necessary steps to design memory-efficient solutions.

One key requirement of our method is a low memory footprint in order to enable parallelization with HTCondor. As a result, using data abstraction to represent the road network is not ideal since it introduces unnecessary overhead. Therefore, our implementation is developed in the C programming language with memory-efficient representations of the road network via C structs and arrays.

The road network can be represented as an adjacency list instead of an adjacency matrix since the network is highly sparse. An adjacency matrix takes $O(|V|^2)$ memory, while



(a) Array of linked lists. (b) Array of arrays.

Figure 3. Adjacency list implementations.

```
typedef struct point_t{
  double lat,lon;
} point_t;
typedef struct edge_t{
  int n1idx, n2idx;
  double len;
  double minlat,minlon, maxlat,maxlon;
} edge_t;
point_t nodearr[max_nodes];
edge_t edgearr[max_edges];
int degarr[max_nodes];
int adjlist[max_nodes][max_degree*2];
```

Figure 4. Road network representation. Variables `max_nodes`, `max_edges`, and `max_degree` are set to $|V|$, $|E|$, and the maximum out-degree of the road network, respectively.

an adjacency list takes $O(|V| + |E|)$ memory [25]. Since the adjacency matrix is highly sparse($|E| << |V|^2$), an adjacency list representation is preferable.

A common implementation of an adjacency list is an array of linked lists as in Figure 3a. However, this implementation requires calling `malloc` for each node in the array and storing pointers between the elements in the linked lists, each of which is 8 bytes assuming IEEE 754 standard double-precision floating point (64 bit). For our problem, the maximum degree of the network can be computed a priori in order to implement the adjacency list as a two-dimensional C array as in Figure 3b. In this way, pointers are not needed and memory is allocated only once.

The road network is defined by `point_t` and `edge_t` structures shown in Figure 4. The full representation consists of four elements: an array of `point_t` structs to store the nodes, and an array of `edge_t` structs to store the edges, an `int` array to store the out-degree of each node, and a two-dimensional array to store the adjacency list. The node and edge arrays allow for constant-time lookup of node and road segment properties, which is necessary for both shortest path computation and feature extraction.

In order to conserve memory, the second dimension of the adjacency list array (the width of the array in Figure 3b, e.g.) is extended by a factor of two. This enables encoding of both the node index and the edge index for each neighbor so that Dijkstra's algorithm can traverse the road network with constant-time lookup of node and road segment properties. The number of neighbors of each node in the adjacency list is encoded in the degree array.

Assuming `double`= 8B, `int`=4B, and 3B compiler

TABLE I: Memory required for road network representation.

Struct name	Memory	Array name	Memory
`point_t`	19B	`nodearr`	1.5MB
`edge_t`	51B	`edgearr`	11MB
		`degarr`	320KB
		`adjlist`	6MB
		Total	**20MB**

```
typedef struct node_t{
  int taxicount, tripcount;
  int taxicount_t[TIMERES];
  int taxicountS_t[TIMERES];
  int taxicountD_t[TIMERES];
  int tripcount_t[TIMERES];
  int tripcountS_t[TIMERES];
  int tripcountD_t[TIMERES];
  double tipfracsum_nt[TIMERES];
  double tipfracsumS_nt[TIMERES];
  double tipfracsumD_nt[TIMERES];
  int passcount_nt[TIMERES];
  int passcountS_nt[TIMERES];
  int passcountD_nt[TIMERES];
} node_t;
```

Figure 5. Example C structure to extract taxi and trip counts, tips, and number of passengers for pickups ("S"), dropoffs ("D"), and along the Dijkstra paths. Variable `TIMERES` refers to the number of time points to track, e.g., 168 for each hour of the week.

padding of C structs, a `point_t` struct takes 19 bytes while an `edge_t` struct takes 51 bytes. As a result, the node array takes 1.5 MB, the edge array is 11 MB, the degree array is 320 KB, and the adjacency list is 6 MB as shown in Table I, for a total of 20 MB. Although the network is too large to be stored as a local variable, it is small enough to be stored as a global variable.

Shortest path computation. The full road network in Figure 4 as well as an additional fixed-length array to store the current path is required for the shortest path computation. The taxi data provides the pick-up and drop-off coordinates to compute the paths. The data takes about 16.7 GB space, which is small enough to be stored on a desktop hard drive but is infeasible to load it at runtime on the available cluster machines, which have 8 GB or 16 GB RAM. For this reason, the data is read line by line as a filestream to compute each shortest path.

Feature extraction. In order to calculate statistics on the taxi movement at each road network node over time, the taxi data needs to be matched with its corresponding shortest path data. This is accomplished by writing the shortest paths to file, and then opening two filestreams to compute statistics.

In addition, computing statistics requires storing counters that track the statistic at each location captured by the

NYC road network. For this purpose, a `node_t` struct is defined as in Figure 5. Assuming `double`=8B and `int`=4B as well as 3B padding, this struct takes 10 MB and the full array takes about 800 MB to store. In practice, more statistics can be tracked simultaneously so that the full array is on the order of 2 or 3 GB. In this way, the statistics computation has a much larger memory footprint than the shortest path computation, which only has the 20 MB road network as a major memory requirement. The difference in memory footprints is one reason to separate the steps into pre- and post-processing jobs to submit to HTCondor.

B. Algorithm Design

The algorithms used for the backbone workflow are presented. The first step is to estimate the taxi trajectory, which is approximated as the path between a trip's start- and end-coordinates that minimizes the distance traveled. Then statistics along the trajectory are computed.

Shortest paths. As discussed in Section II, the memory constraints of the available cluster and the road network properties (sparse and directed with non-negative edge weights) lead to the choice of Dijkstra's algorithm for the implementation, with the Euclidean distance of road segments $e \in E$ as the weights.

The min-priority queue implementation of Dijkstra's algorithm may be implemented in $O(|V|^2+|E|) = O(|V|^2)$ time [25]. Figure 6 shows the inner loop of the algorithm. The runtime depends on the implementation of the min-priority queue and has been shown to be faster with binary heaps or Fibonacci heaps [25], [32]. Since the road network is sparse, binary heap implementations can run in $O(|E|\log|V|)$ time on a strongly connected graph, assuming $|E| = o(|V|^2/\log(|V|))$ [25]. A Fibonacci heap implementation runs in $O(|E|+|V|\log|V|)$ time [25]. Our implementation implements binary heaps since they have been shown to perform better in practice [32], [35].

We analyze the computational requirement for the shortest path algorithm in terms of floating point operations per second (FLOPS). Since the NYC road network is sparse, Dijkstra's algorithm is expected to run in $O(|E|\log|V|)$ time, which then requires about $300\text{K}\cdot\log(79\text{K})$ or 1.5M floating-point operations. For a single core of an Intel Core i5-6500T processor using x87 scalar code (2 flops per cycle at 2.5 GHz, or 5 GFLOPS), the corresponding runtime is about 300μs. Running 700 million computations requires about 10^{15} floating point operations for a runtime of about 57 hours or 2.4 days. Assuming 20% of peak performance (1 GFLOPS), the 700 million computations would run in about 290 hours, or 12 days.

Feature extraction. On the other hand, extracting statistics from a taxi trajectory is approximately linear as $O(|V|)$ since the computation is, to first order, a constant-time operation at each node of the shortest path. Scaling to the 700 million trips, the theoretical runtime on the same Core i5-6500T 2.5 GHz processor using x87 scalar code is about 3 hours, or 15 hours assuming 20% peak performance.

```
for (i=0; i<degarr[v]; i++){
  w = adjlist[v][i*2];
  e = adjlist[v][i*2 + 1];
  currdist = dist[v] + edgearr[e].len;
  if (!visited[w] || currdist < dist[w]){
    dist[w] = currdist;
    endq = (endq + 1) % numnodes;
    insert_by_priority(distQ,Q,currdist,w);
    paths[w] = v;
}}
```

Figure 6. Dijkstra inner loop at node `v` showing min-priority queue implementation with arrays `Q` and `distQ`.

Compared to the 12 days needed to run the Dijkstra algorithm, the computation time of the statistics is relatively short. We expect to run Dijkstra's algorithm once assuming the underlying road network is stable over time, while the post-processing step can be repeated many times as new research questions necessitate further queries. In this way, splitting the workflow into a pre-processing step consisting of the Dijkstra computations and a post-processing step for the statistics computations is appropriate for knowledge discovery for the NYC taxi data.

To summarize, an analysis that justifies separating the solution workflow into pre- and post-processing steps is presented. Pre-processing requires loading the 20 MB road network and computing Dijkstra's algorithm with expected runtime of 12 days for 700 million computations. The post-processing requires loading a 2 or 3 GB struct array in addition to the road network but can complete in 15 hours.

Section III-C next discusses how to shorten the 12-day shortest path computation and 15-hour feature extraction by exploiting the high-throughput nature of the problem.

C. High Throughput Computing

The path and feature computations fit a high-throughput paradigm since they can be separated into independent jobs. Sections III-B and III-A show that the Dijkstra implementation computes a single taxi path in 300μs assuming reasonable memory resources. Therefore, the path computations can be run in parallel in order to shorten the 12-day expected runtime.

Our parallelized computation utilizes HTCondor, an open-source tool that provides a high throughput computing environment [15]. A user submits a series of jobs to HTCondor, which waits until a machine on the dedicated cluster is idle to start a job. The cluster we use has 32 machines that are either 16-core, 16 GB RAM or 8-core, 8 GB RAM. Assuming 8-core machines with the same Core i5 processors in Section III-B, the 57 CPU-hour estimate can be reduced to about 0.2 wall clock hours; at 20% peak performance, an HTCondor implementation is expected to

reduce the runtime from 290 CPU-hours (12 days) to 1 or 2 wall clock hours. Note that these projections do not account for overhead such as the time to load the road network into RAM, which makes high-throughput computing even more essential for reducing the time to solution. This analysis shows the impact of exploiting the high-throughput nature of the workflow.

The design decisions described in this section are crucial to attain a workable solution because of the large scale of the research problem. The primary design decisions for memory efficiency have been discussed. The implementation choices for Dijkstra's algorithm and its expected runtime have been described. In addition, the impact of implementing a high-throughput solution for reducing the time to solution has been analyzed. The next section provides the high-level steps in the solution workflow.

IV. Implementation

We provide the implementation of the code workflow. Section IV-A describes the parallelization of the Dijkstra and statistics computations with HTCondor. Sections IV-B and IV-C detail the steps and additional design decisions for the shortest path computation and feature extraction implementations.

A. Parallelization

The overall workflow of the solution is the following:

```
1: function MAIN
2:     SHORTEST PATH COMPUTATION
3:     FEATURE EXTRACTION
4:     AVERAGING
5: end function
```

Lines 2 and 3 are separate high-throughput problems that can be run in HTCondor. Here the steps to run the shortest path computations and feature extraction as jobs on HTCondor are described. The job submission requirements, the number of jobs to run per machine, and handling input and output are described.

Shortest paths on HTCondor. The first step is to determine the requirements of the cluster machines in an HTCondor ClassAd. Since HTCondor writes the output as a text file to the scheduled machine, machines with enough physical memory to store uncompressed output are required. Assuming that a job computes 500,000 shortest paths, the output file requires at most 3 GB of physical memory. In addition, machines that are either 32-bit or 64-bit with a Linux operating system are requested. A sample ClassAd is shown in Figure 7.

The cluster has 32 machines that are either 16-core, 16 GB RAM or 8-core, 8 GB RAM, so there are 300 to 500 available cores depending on the demand and the machines that HTCondor chooses for scheduling. However, since the HTCondor output is written to the submit machine, the output needs to be compressed and transferred to a local machine. The submit machine and local machine both have limited physical memory, so it is not possible to compute all 700 million shortest paths before compressing the data. In addition, the machine cannot compress many files at once since the process consumes RAM and slows down the cluster. Since fewer shortest paths per job decreases execution time per job but increases the number of jobs, there is a trade-off between the paths computed per job and the execution time on the high throughput platform.

```
Requirements = (Arch == \"INTEL\"
                || Arch == \"X86_64\")
                && OpSys == \"LINUX\"
Request_Memory = 3G
```

Figure 7. Example HTCondor ClassAd for shortest path computation.

To account for these issues, each job is assigned to about 500,000 shortest paths, which takes about an hour to complete (see Table II). The total computation for 700 million taxi trips requires 1,500 jobs. Since 30 CSV files of size 3 GB can be compressed at a time without noticeably slowing the cluster, 30 jobs are submitted at a time with shell scripts that monitor the status of the HTCondor jobs and start output compression once a job completes. After half of the original jobs are complete, another set of 30 jobs is added to the batch queue.

Feature extraction on HTCondor. A `node_t` struct for feature extraction as specified in Figure 5 takes about 2 GB of memory assuming 64-bit (see Section III-A). For this reason, the ClassAd of Figure 7 is modified to account for this memory requirement.

The feature extraction runs over the shortest path results so that each extraction job corresponds to one shortest path file. As in the shortest path computation, 30 jobs are submitted at a time. A new batch of 30 jobs is submitted every 10 minutes since the each job runs in 10-15 minutes.

The design considerations for implementing Dijkstra's algorithm and feature extraction in C are detailed below. Section IV-B presents the pre-processing solution. Section IV-C presents the post-processing solution.

B. Pre-processing implementation

The pre-processing workflow is as follows:

```
1: function SHORTEST PATH COMPUTATION
2:     Define NYC map.
3:     Load NYC road network.
4:     Open taxi data filestream.
5:     for each line in taxi data do
6:         Check for errors.
7:         Match coordinates to map.
8:         Compute shortest path.
9:         Write path to file.
10:    end for
11: end function
```

These steps are presented in detail below.

Defining the New York City map. The region of New York City is modeled as a union of rectangular bounding boxes that are defined by the top-left, top-right, bottom-left, and bottom-right coordinates. Other geography representations include ESRI Shapefiles [36], which have the benefit of providing more detailed models for geographic boundaries such as coastlines and city boundaries. OpenStreetMaps [37] represents geographical areas as points, lines and polygons in Geographic Data Files. We choose to represent the geography with the rough approximation provided by an array of bounding boxes since it takes less time to load at runtime.

Loading the road network. As discussed in Section III-A, the shortest path implementation loads the road network at runtime. The network is represented in terms of an adjacency list as described in Section III-A and illustrated in Figures 3b and 4. Additional preprocessing of the network such as sorting the array of nodes `nodearr` in Figure 4 facilitates mapping the taxi data to the NYC map, as discussed later in this section.

Reading the taxi data. A single line of the NYC taxi data is parsed to obtain the start- and end-coordinates of a taxi trip as input to the Dijkstra algorithm.

Handling data errors. One important issue is the presence of errors in the NYC taxi data. These errors appear as invalid geo-coordinates and timestamps as well as invalid trip distances and durations. For example, certain geo-coordinates lie in the middle of Hudson River or on top of the Empire State building. We develop a set of criteria to determine whether a trip is spurious or not.

Our error checking works as follows. The start- and end-coordinates of a trip are first verified to be contained in the New York City bounding boxes as defined above. Trips with geo-locations that lie outside these boxes are discarded. Other GPS errors include zero geo-coordinates and trip distances that are reported as less than the Euclidean distance between the start and end points. Trips of duration less than a minute are discarded, as well as trips of distance less than 0.2 miles since the road network resolution is such that each road segment is at least 0.2 miles long.

If both coordinates of a taxi trip satisfy the error-checking conditions, they are mapped to the road network. The map-matching method is described next.

Map matching. The geo-coordinates in the taxi data are mapped to the road network coordinates stored in `nodearr` in Figure 4 with a map-matching technique known as perpendicular mapping, or nearest-point estimation. Other map-matching methods are discussed in [38], [39], [40]. This map-matching method finds the "closest" point in the road network by computing the orthogonal distance to the road segments in the road network. The initial step involves finding a small subset $V_s \subset V$ of nodes with the same latitude as the given coordinate; using binary search on the sorted node array, this takes $O(\log |V|))$ time.

The next step is to compute the orthogonal distance from the geo-coordinate to each road segment that has at least one endpoint contained in the node subset; i.e., the distance is computed for all $(v_i, v_j) \in E$ such that $v_i \in V_s$ or $v_j \in V_s$. If (v_i^*, v_j^*) is the road segment that minimizes the distance, then the coordinate from the taxi data is mapped to the closest endpoint. Map matching is performed for both the pick-up coordinate and the drop-off coordinate.

In some cases, the distance between the original coordinate and the matched coordinate is very large. The trip is discarded if this distance is greater than 0.2 miles for either the start- or the end-coordinate.

C. Post-processing implementation

In this section, the high-level implementation for feature extraction from the NYC taxi data is described. The overall method is as follows:

```
1: function FEATURE EXTRACTION
2:     Update node_t.
3:     Define NYC map.
4:     Load NYC road network.
5:     Open taxi data and shortest path filestreams.
6:     for each line in taxi data do
7:         Check for errors.
8:         Match coordinates to map.
9:         Sync line to shortest path data.
10:        Compute statistics.
11:        Write statistics to file.
12:    end for
13: end function
```

Updating `node_t`. The node struct is updated to include statistics of interest, such as total number of trips, total number of passenger, and total fare paid. These statistics are defined as arrays with length corresponding to the time resolution we desire. Weekly averages are computed and stored in 168-element arrays such that index 0 corresponds to Sunday 12am-1am, index 1 corresponds to Sunday 1am-2am, index 24 corresponds to Monday 12am-1am, etc.

Error checking and syncing filestreams. As in the preprocessing step, reading the original data requires defining the NYC map and road network as well as implementing identical error checks. In this way, a valid line in the preprocessing step can be matched to its computed path in the corresponding shortest path data file.

Algorithm. For each valid trip, the data fields of interest are first extracted. Then, for each node on the shortest path, the fields in the corresponding `node_t` struct are updated to reflect the values from the taxi data. For a data-specific value such as total fare, the value from the data is extracted and used to update the corresponding node struct value. For updating a trip count, the corresponding counter is incremented in the node struct. These operations are constant-time operations but have a large constant because of the large number of paths to process.

TABLE II: Speedup across platforms for shortest path computations (pre-processing).

Platform	No. trips per hour	Total runtime (in hours)	Speedup vs. Python	Speedup vs. C
Python	10,000	72,000		
C	500,000	1,440		50x
C + HTCondor	500,000 (1 job)	6.5 (1,584 jobs)	11,000x	220x

It is straightforward to modify the operations to investigate other taxi features. The simplicity of these operations is important in order to extract statistics quickly.

Writing output and computing averages. The output is written to a CSV file through HTCondor as described in Section IV-A. Converting the totals to average statistics requires one more pass over the files containing the statistics. The time to solution is described in Section V.

Our post-processing step is easy to modify for research purposes. For example, the `node_t` struct can be updated to track different statistics. Furthermore, the set of operations in the computation algorithm can be changed. This step is designed to be streamlined for fast feature extraction to enable analysis of taxi movement.

This section and Section III have shown that a workable solution can be achieved by separating the workflow into pre- and post-processing steps that are each run in a high-throughput environment. This design decision allows for flexibility in the choice of taxi-based features to extract from the data. The speed of the two-step solution is presented in the next section.

V. Experimental Results

This section compares our design over several frameworks in terms of relative speedup. A two-step solution is implemented in C and HTCondor to compute 700 million shortest paths and to extract statistics such as average trip counts from 2010-2013 New York City taxi data. Results for the shortest path computations are first presented, and then the feature extraction step is discussed.

Pre-processing. Table II summarizes the speedup results for the pre-processing step. The initial run of this algorithm was done in Python on a 64-bit machine with an Intel Core i5-4300U CPU at 2.50 GHz. Computing the first 10,000 shortest paths took about one hour. Each month contains about 15 million trips, which would take about 1,500 hours to compute. There are 48 months in the entire data set, so the total number of trips to compute is on the order of $48 \cdot 1,500 = 72,000$ hours, or 3,000 days. The current implementation in C returns shortest paths for 500,000 trips in 1 hour, which corresponds to 30 hours for 1 month, or $30 \cdot 48 = 1440$ hours or 60 days to get shortest paths for the entire data set, which is 50x speedup compared to the Python implementation. This involved using the memory and latency design outlined in Section III.

The execution time is further reduced by leveraging the high-throughput environment of HTCondor to submit simultaneous jobs to a 32-machine cluster that are either 16-core, 16 GB RAM or 8-core, 8 GB RAM. Each job computes 500,000 paths, for a total of 1,584 jobs. The shortest paths for the 700 million trips were computed in 6.5 hours, which is a speedup of about 220 compared to non-simultaneous performance of 60 days and a speedup of about 11,000 compared to the original Python implementation.

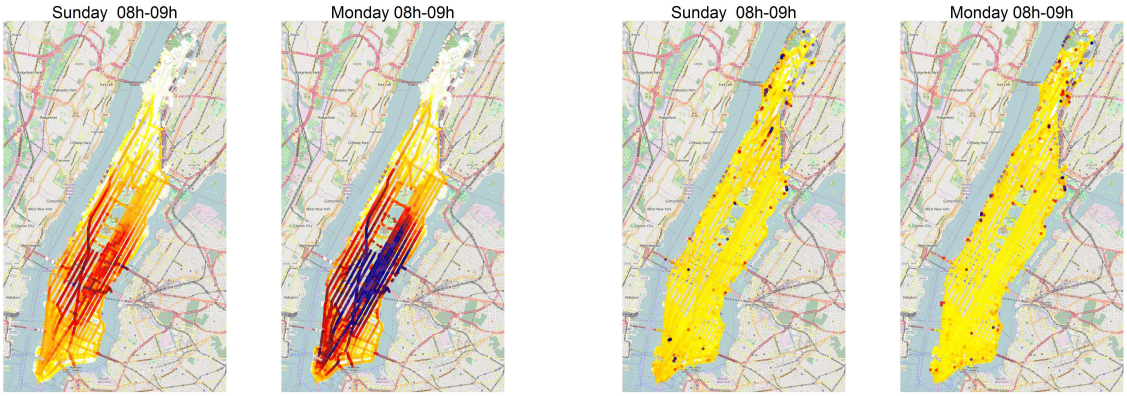
Post-processing. The `node_t` struct in Figure 5 is designed to compute taxi counts, trip counts, tip, passenger counts, return frequencies and return times for the paths as well as pick-ups and drop-offs. The time resolution was chosen to be each hour of the week, so the struct took about 2 GB in memory.

The C implementation for feature extraction computes 175 million trip statistics (1 year of taxi data) in one hour, or 700 million trip statistics in four hours. Each year is run as a separate job on HTCondor so that the total statistics were computed in 1 hour. Averaging required an additional 4 hours, for a total runtime of 5 hours for post-processing. Figure 8 shows some example results.

In total, pre- and post-processing together have a runtime of 11.5 hours, or about half a day. Compared to the single-machine Python implementation with projected runtime of 3,000 days, a workable solution is achieved that can be run and rerun to study and evaluate taxi statistics on New York City.

VI. Discussion

Using the design principles discussed in this paper, statistics that characterize taxi movement on the New York City road network can be extracted and explored. Two example statistics are the average number of trips and the average number of passengers per trip at each intersection of the Manhattan grid. Figure 8a shows that the average number of taxis passing through Manhattan is much higher 8am–9am on Mondays compared to 8am–9am on Sundays, which reflects the expected weekday rush hour congestion. Figure 8b shows that taxis that pick up more passengers tend to have trajectories around the perimeter via Hudson River Parkway or FDR Drive instead of through the heart of Manhattan. We also observe that the average number of passengers per trip decreases on Mondays for trips that travel through or near the Brooklyn Bridge. Such statistics can be analyzed with frameworks such as tensor analysis [42], [43] and graph signal processing [7], [44], for example, in order



(a) Average number of trips

(b) Average number of passengers per trip

Figure 8. Four-year average June-August statistics on Manhattan, NYC, for Sundays and Mondays 8am to 9am. Colors denote $\log_{10}$ bins of (a) the average number of trips (699 log bins; white to yellow: 0–12, orange: 12–92, red: 92–320, blue: 320–280, purple: 280–880, black: 880–1,430) and (b) the average number of passengers per trip (499 log bins; white to yellow: 0–1.6, orange: 1.6–2.1, red: 2.1–3.4, blue:3.4–4.2, purple:4.2–4.9, black: 4.9–6). The plots were generated with ggmap [41] and OpenStreetMap [37].

to conduct fine-grained spatiotemporal analyses of taxi flows in New York City.

Such observations and subsequent analysis are highly dependent on the route-finding algorithm, which is Dijkstra's algorithm with road network distances as the weights (see Section III-B). This particular choice explains certain features in Figure 8, such as the high number of trips on Broadway in Figure 8a. A possible solution is to modify the weights of Dijkstra's algorithm; for example, a weighting scheme that would disperse the concentration of taxi trips is achieved by defining for each edge $(v_i, v_j) \in E$ the weight $w_{ij} = \alpha d_{ij} + (1-\alpha) n_{ij}$, where $\alpha \in (0,1)$, d_{ij} is the (normalized) Euclidean distance between v_i and v_j, and n_{ij} is the (normalized) average number of trips from v_i to v_j. Since n_{ij} is an output of our computation, we can run multiple iterations such that the output $n_{ij}^{(p)}$ of iteration p is the input to the weights of iteration $p+1$. This may continue until the process converges (e.g., $\left\| n_{ij}^{(p+1)} - n_{ij}^{(p)} \right\| < \epsilon$ for all i, j and threshold $\epsilon > 0$). The applicability of such an iterative method provides additional motivation for the development of fast, efficient solutions to compute statistics.

Furthermore, different Dijkstra cost functions must be implemented to address different questions. For example, historical congestion information for each NYC road, capacity of each road measured in terms of the number of lanes, and the expected number of pedestrians that stop traffic at each intersection at a particular time of day would provide invaluable information as to the optimal trajectory for a taxi to take through the city. In addition, if the problem constraints change, such as defining optimality in terms of the route that minimizes air pollution as addressed in [45], the Dijkstra cost function should change to address the problem.

VII. Conclusion

We provide a solution to the problem of studying taxi movement in New York City that runs in less than a day. Using four years of historical taxi data, static representations of the taxi trips are converted to dynamic representations via Dijkstra's algorithm. These paths are used to extract statistics, such as average trip counts, that reflect taxi flows through the city. Our contribution is to show the details of reducing the time-to-solution from 3,000 days to less than a day. The latency and memory costs for the problem are described in detail and motivate our C implementation. Moreover, considerations for making the problem parallelizable for HTCondor are discussed. Our solution enables traffic studies on the New York City road network that take into account 700 million taxi trips.

References

[1] N. Ferreira, J. Poco, H.T. Vo, J. Freire, and C.T. Silva, "Visual exploration of big spatio-temporal urban data: A study of New York City taxi trips," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 12, pp. 2149–2158, 2013.

[2] B. Donovan and D.B. Work, "Using coarse GPS data to quantify city-scale transportation system resilience to extreme events," *presented at Transp. Res. Board 94th Annual Meeting (arXiv:1507.06011 [physics.soc-ph])*, Jan. 2015.

[3] "NYC Open Data," https://nycopendata.socrata.com/.

[4] New York City Dept. of Transportation, "Real time traffic information," URL: http://dotsignals.org/, Accessed 25-May-2016.

[5] B. Donovan and D.B. Work, "New York City Taxi Data (2010-2013)," Dataset, http://dx.doi.org/10.13012/J8PN93H8, 2014, Accessed 30-Jun.-2016.

[6] D. Chu, D. A. Sheets, Y. Zhao, Y. Wu, J. Yang, M. Zheng, and G. Chen, "Visualizing hidden themes of taxi movement with semantic transformation," in *2014 IEEE Pacific Visualization Symp.*, Mar. 2014, pp. 137–144.

[7] J.A. Deri and J.M.F. Moura, "Taxis in New York City: A network perspective," in *Proc. 49th Asilomar Conf. Signals, Syst., and Comput.*, Nov. 2015, pp. 1829–1833.

[8] C. Chen, D. Zhang, P.S. Castro, N. Li, L. Sun, and S. Li, "Real-time detection of anomalous taxi trajectories from GPS traces," in *Mobile and Ubiquitous Syst.: Computing, Networking, and Services*, pp. 63–74. Springer, 2012.

[9] X. He and H.X. Liu, "Modeling the day-to-day traffic evolution process after an unexpected network disruption," *Transp. Res. Part B: Methodological*, vol. 46, no. 1, pp. 50–71, 2012.

[10] P.S. Castro, D. Zhang, and S. Li, "Urban traffic modelling and prediction using large scale taxi GPS traces," in *Pervasive Computing*, pp. 57–72. Springer, 2012.

[11] Y. Zheng, Y. Liu, J. Yuan, and X. Xie, "Urban computing with taxicabs," in *Proc. 13th Int. Conf. Ubiquitous Computing*. ACM, 2011, pp. 89–98.

[12] R. Puzis, Y. Altshuler, Y. Elovici, S. Bekhor, Y. Shiftan, and A. Pentland, "Augmented betweenness centrality for environmentally aware traffic monitoring in transportation networks," *J. Intell. Transp. Syst.*, vol. 17, no. 1, pp. 91–105, 2013.

[13] B. Furht and A. Escalante, *Handbook of Data Intensive Computing*, New York, NY, USA: Springer Science & Business Media, 2011.

[14] Google, "Google Map of New York, New York," https://goo.gl/maps/57U3mPQcdQ92, Accessed 31-Aug.-2014.

[15] HTCondor, *version 8.2.10*, Center for High Throughput Computing, Univ. of Wisconsin-Madison, 2015.

[16] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *Concurrency – Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[17] Python, *version 2.7*, Python Software Foundation, 2015.

[18] MATLAB, *version 8.5.0 (R2015a)*, The MathWorks Inc., Natick, Massachusetts, 2015.

[19] J. Bezanson, S. Karpinski, V.B. Shah, and A. Edelman, "Julia: A Fast Dynamic Language for Technical Computing," *arXiv:1209.5145 [cs.PL]*, Sep. 2012.

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th USENIX Symp. Operating Syst. Design and Implementation*, 2004, p. 10.

[21] Hadoop, *version 2.6.4*, Apache Software Foundation, 2015.

[22] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing*, 2010, p. 10.

[23] R.S. Xin, J.E. Gonzalez, M.J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *1st ACM Int. Workshop Graph Data Management Experiences and Syst. (GRADES)*, 2013, pp. 2:1–2:6.

[24] E.W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[25] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson, *Introduction to Algorithms*, McGraw-Hill, 2nd edition, 2001.

[26] S. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," in *Proc. 15th Int. Conf. Parallel Architectures and Compilation Techn. (PACT)*, 2006, pp. 222–232.

[27] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering route planning algorithms," in *Algorithmics of Large and Complex Networks*, pp. 117–139. Springer, 2009.

[28] A.V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proc. 16th Annual ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2005, pp. 156–165.

[29] C. Sommer, "Shortest-path queries in static networks," *ACM Computing Surveys*, vol. 46, no. 4, pp. 45:1–45:31, Apr. 2014.

[30] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Experimental Algorithms*, vol. 5038, pp. 319–333. Springer Heidelberg, 2008.

[31] J. Zhang, "Efficient frequent sequence mining on taxi trip records using road network shortcuts," in *Big Data: Techniques and Technologies in Geoinformatics*, H.A. Karimi, Ed., pp. 193–206. CRC Press, 2014.

[32] A.V. Goldberg and R.E. Tarjan, "Expected performance of Dijkstra's shortest path algorithm," *NEC Res. Inst. Rep.*, Jun. 1996.

[33] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," in *Proc. 35th Int. Conv. Inform. and Communication Technol., Electronics and Microelectronics (MIPRO)*, May 2012, pp. 1811–1815.

[34] Baruch College: Baruch Geoportal, "NYC Geodatabase," URL: https://www.baruch.cuny.edu/confluence/display/geoportal/NYC+Geodatabase, Accessed 31-Aug.-2014.

[35] B.V. Cherkassky, A.V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Math. Programming*, vol. 73, no. 2, pp. 129–174, 1996.

[36] ESRI, "ESRI Shapefile Technical Description," Tech. Rep., ESRI, Jul. 1998.

[37] OpenStreetMap contributers, "OpenStreetMap," http://www.openstreetmap.org/, Accessed 09-Jun.-2016.

[38] F.C. Pereira, H. Costa, and N.M. Pereira, "An off-line map-matching algorithm for incomplete map databases," *Eur. Transport Res. Rev.*, vol. 1, no. 3, pp. 107–124, 2009.

[39] M.A. Quddus, W.Y. Ochieng, L. Zhao, and R.B. Noland, "A general map matching algorithm for transport telematics applications," *GPS Solutions*, vol. 7, no. 3, pp. 157–167, 2003.

[40] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk, "On map-matching vehicle tracking data," in *Proc. 31st Int. Conf. Very Large Data Bases (VLDB)*, 2005, pp. 853–864.

[41] D. Kahle and H. Wickham, "ggmap: Spatial visualization with ggplot2," *The R Journal*, vol. 5, no. 1, pp. 144–161, 2013.

[42] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: Dynamic tensor analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2006, pp. 374–383.

[43] K. Maruhashi, F. Guo, and C. Faloutsos, "MultiAspectForensics: Pattern mining on large-scale heterogeneous networks with tensor analysis," in *Proc. Int. Conf. Advances in Social Networks Analysis and Mining (ASONAM)*, Jul. 2011, pp. 203–210.

[44] A. Sandryhaila and J.M.F. Moura, "Big data analysis with signal processing on graphs: Representation and processing of massive data sets with irregular structure," *IEEE Signal Process. Mag.*, vol. 31, no. 5, pp. 80–90, Aug. 2014.

[45] M.H. Sharker and H.A. Karimi, "Computing least air pollution exposure routes," *Int. J. Geographical Inform. Sci.*, vol. 28, no. 2, pp. 343–362, 2014.