Urban Data Integration Using Proximity Relationship Learning for Design, Management, and Operations of Sustainable Urban Systems

Karan Gupta[1]; Zheng Yang, Ph.D., A.M.ASCE[2]; and Rishee K. Jain, Ph.D., A.M.ASCE[3]

Abstract: The world is rapidly urbanizing, with 66% of the world's population expected to reside in cities by 2050. This massive influx of new urban citizens is putting enormous pressure on city systems and bringing forth challenges at the intersection of urban infrastructure, governance, and the environment. As a result, researchers and practitioners have turned to new advanced sensing and data analytics developed under the burgeoning smart city movement to improve the design, management, and operations of urban systems. However, it has been challenging to integrate, organize, and analyze the data emerging from urban systems due to their natural spatial, temporal and typological heterogeneity. This paper introduces an urban data integration (UDI) framework that is capable of integrating heterogeneous urban data. The proposed UDI framework is extensible to multiple types of urban systems, scalable to the growing volume of data streams (as a result of increasing geographical areas, higher sampling frequencies, and so on), and interpretable enough to help inform municipal decision-making. The UDI framework uses a series of proximity relationship learning algorithms to reconstruct urban data in a graph database. The merits, applicability, and efficacy of the proposed framework is demonstrated by validating and testing it on data from a midsize city in the United States and by benchmarking its interpretability and computational performance for a typical urban analytics scenario against current practice (i.e., a relational database). Results indicate that the UDI framework provides easier and more computationally efficient exploration and querying of urban data, and in turn can enable new computational approaches to urban system design, management, and operations. **DOI: 10.1061/(ASCE)CP.1943-5487.0000806.** © *2018 American Society of Civil Engineers.*

Author keywords: Data integration; Graph database; Proximity learning; Smart city; Urban data.

Introduction

The world is experiencing rapid urban growth. Over 50% of world's population now resides in cities, with this number expected to increase to 66% (i.e., 2.5 billion additional people) by 2050 (United Nations 2014). This rapid urban growth has begun to significantly increase the demands on urban systems and is in-turn creating numerous challenges at the intersection of urban infrastructure, governance, and environment. In particular, municipal decision makers are grappling with the need to better design, manage, and operate urban systems so that they can meet the demands of their citizens, provide equitable access to core services, and limit negative impacts on the environment. The urban environment has a complex network of interconnected systems that have become increasingly challenging to manage using existing management paradigms (IBM 2012). Changes in one system can have substantial (nonlinear) impacts on another system, making it difficult to discern and predict the effects of urban design, management, and policy decisions. Moreover, numerous interdependencies exist between various urban systems, including building–transportation (Marique et al. 2017), building–human (Langevin et al. 2015), building–vegetation (Perini and Magliocco 2014), transportation–land, and environment–human–transportation interdependencies (Raymond et al. 2013). For example, transportation planning should not only consider the reduction of congestion but also the minimization of impacts on local air quality and building heating/cooling loads. Data-driven approaches could discover the sources and context of such problems, and decompose the interconnected aspects into tasks that enable proactive analysis and operational decision-making. As part of the burgeoning smart city movement, massive amounts of data are now being collected on an array of urban systems (e.g., land, vegetation, buildings, transportation, energy, and humans), which in turn provides a tremendous opportunity to leverage emerging data-driven methods to facilitate the sustainable planning, management, and operation of urban systems.

Municipal officials, policy makers, and engineers are eager to adopt more data analytical approaches to uncover insights into how their cities operate and drive decision-making. However, they often run into several challenges related to data integration, organization, and analysis due to the natural spatial, temporal, and typological heterogeneity of urban data. Urban data streams can differ significantly in their spatial resolution (e.g., building, community, and urban), time scale (e.g., hourly, monthly, and yearly), and typological representation (e.g., categorical, numerical, and geometric). Additionally, due to the disparate nature of urban systems, data streams are often used for domain-specific analysis even if applicable to multiple systems and are limited in their accounting

[1]Research Assistant, Urban Informatics Laboratory, Dept. of Civil and Environmental Engineering, Stanford Univ., 473 Via Ortega, Room 269B, Stanford, CA 94305.

[2]Staff Researcher, Urban Informatics Laboratory, Dept. of Civil and Environmental Engineering, Stanford Univ., 473 Via Ortega, Room 269B, Stanford, CA 94305.

[3]Assistant Professor, Urban Informatics Laboratory, Dept. of Civil and Environmental Engineering, Stanford Univ., 473 Via Ortega, Room 269A, Stanford, CA 94305 (corresponding author). Email: rishee.jain@stanford.edu

Note. This manuscript was submitted on February 21, 2018; approved on August 3, 2018; published online on December 13, 2018. Discussion period open until May 13, 2019; separate discussions must be submitted for individual papers. This paper is part of the ***Journal of Computing in Civil Engineering***, © ASCE, ISSN 0887-3801.

for interdependency and interactions between systems. For example, building energy data are useful for diagnosing energy efficiency performance of a city but they could also be used for smart grid planning and understanding urban heat island dynamics. Building energy systems also interact or have interdependency with other systems in a city including natural (e.g., trees), transport, and human systems. Failing to account for such interdependencies could result in decision-making that narrowly focuses on the benefits of a single system (e.g., buildings) without understanding potential impacts on other systems. This paper defines interdependency as the mutual influences between urban elements (e.g., the transportation system can impact building heating/cooling loads, and building occupants can impact traffic congestion). Interaction is defined as the reciprocal actions among urban elements (e.g., humans control building systems and building heating/cooling change human comfort and schedules). An urban system (e.g., building system) is defined as a collection of urban elements (e.g., buildings), and a domain is defined as the design, management, and operation of a specific urban system (e.g., energy, transportation, or water). Lastly, this paper defines the term proximity relationship as the adjacency property of two urban elements (e.g., whether/how much they are adjacent to each other) in the denoted study area (e.g., building, road, or tree).

The heterogeneity of urban data and the nascent field of urban analytics both point toward the need for integrating urban data early and often. For example, if new data become available on the air quality along a main traffic corridor, a municipal official could want to map this new information to existing data sources on related systems (e.g., traffic and roads), rerun analytical queries to understand mutual influences, and then take appropriate actions. Maintaining a high level of interpretability is vital during the integration process because the goal is to support urban design and operational decisions by municipal officials, policy makers, and engineers. As a result, a useful urban data integration framework must be extensible to multiple urban systems (and not system specific), scalable to the growing amounts of quickly changing urban data streams, and interpretable so that it can inform decision-making.

This paper introduces an urban data integration (UDI) framework that integrates heterogeneous urban data while maintaining extensibility, scalability and interpretability. The proposed UDI framework uses a series of proximity relationship learning algorithms to automatically reconstruct urban data in a graph database that can be efficiently and easily explored and queried by municipal officials, policy makers, and engineers. This paper is organized as follows. Section "Related Work" describes related work and demonstrates the gaps in the current literature. Section "UDI Framework" introduces the methodology of the UDI framework and the underlying algorithms in detail. Section "Case Study: Palo Alto, California" demonstrates the merits, applicability, and efficacy of the proposed framework by validating and testing it on a sample of test data from a midsize city in the United States (Palo Alto, California); section "Case Study: Palo Alto, California" also benchmarks UDI's interpretability and computational performance for a typical urban analytics scenario against an existing data management method (i.e., relational databases). Section "Limitations and Future Work" discusses the limitations and potential areas of future work, and section "Conclusions" concludes the paper. Although numerous interdependencies exist across various urban systems, the focus of this paper is to integrate heterogeneous urban systems data based on geographic interdependencies. We use geographic interdependencies as a basis for our integration because this provides a natural first step toward unbiased and non-task-specific data integration while maintaining extensibility, scalability, and interpretability.

Related Work

Urban data can dramatically differ in spatial resolution, time scale, and type (Balaji et al. 2016; Khan et al. 2013; Chang et al. 2014; Yuan et al. 2012; Zielstra et al. 2013); Table 1 gives examples of urban data heterogeneity. This heterogeneity can be attributed to the varying kinds of urban systems that make up a city (e.g., buildings, land, roads, transport, vegetation, and humans), the diverse sources from which data are acquired (e.g., municipal records, in situ sensors, surveys, and ad hoc databases), and the degree to which data are spatially distributed (e.g., building level, community level, and urban level) and temporally updated (e.g., subhourly, hourly, daily, monthly, and yearly) (Aljumaily et al. 2017; Calabrese et al. 2015; Wang and Taylor 2015). As a result, an urban data integration framework must first and foremost be able to reconcile the spatial, temporal, and typological heterogeneity that characterizes urban data streams in order to extract insights across urban systems valuable for holistic municipal planning and decision-making.

Numerous data integration methods have been proposed in the literature to address some of the challenges of urban data management. The following sections review the three main categories of existing data integration methods (domain-centric, data-centric, and demand-centric) to discern key limitations and potential research gaps.

Domain-Centric Integration Methods

Domain-centric integration methods aim to exploit the knowledge or structure of a specific system by integrating various data streams. For example, previous work has used network-based methods to represent urban elements as nodes and interactions between

Table 1. Examples of urban data heterogeneity

Urban system	Typological representation	Time scale	Spatial resolution
Buildings	Geometric, location, numeric, categorical attributes, network structure	Subhourly, hourly, monthly, yearly	Building, subbuilding
Land	Geometric, location, survey data, numeric, categorical attributes	Yearly	Community
Roads	Geometric, location, numeric, categorical attributes, network structure	Monthly, yearly	Community, urban
Transportation	Geometric, location, numeric, categorical attributes	Subhourly, hourly, daily	Community, urban
Vegetation	Location, numeric, categorical attributes, survey data	Yearly	Community, urban
Utilities	Geometric, location, numeric, categorical attributes, network structure	Subhourly, hourly, daily, monthly	Building, urban
Environment	Location, numeric, categorical attributes	Subhourly, hourly, daily	Urban
Human	Location, numeric, social network structure	Daily, monthly, yearly	Building, community, urban

elements as edges (Sun and Han 2012; Jain et al. 2014; Wang and Taylor 2015). However, network-based methods are largely constrained to single system modeling (e.g., transportation network or social network) because the relationships between nodes and edges are explicitly specified by domain knowledge and context. Recent research has begun to integrate data beyond a single system, such as the integration of building information modeling (BIM) data and GIS data (Kang and Hong 2013). However, such frameworks frequently encounter limitations when their use is extended to the context of other urban systems (e.g., human systems). Other urban data integration research has focused on ontology-based methods. These methods typically embed the knowledge and available data of a specific domain in an ontology and use the ontology to link various data streams to one another (Yang et al. 2017). Although a certain level of interpretability could be maintained, ontology-based approaches are constrained to specific systems such as buildings (Balaji et al. 2016) or transportation (Seedah et al. 2015) and have a limited ability to describe the complex relationships and interactions between urban systems that have been uncovered in previous work (Langevin et al. 2015; Marique et al. 2017). For example, recent work has demonstrated a strong link between transportation demand and energy consumption in buildings (Karan et al. 2016). Moreover, a limitation of ontology-based data integration methods is that they must be fully expressive when defining their metadata schema and thus cannot adapt easily to new urban data streams and types (Sinnott et al. 2012; Zhu and Ferreira 2015). Although domain-centric methods are effective for data integration within individual system domains, they are limited in their ability to both account for interactions across systems (extensibility) and easily adapt to growing and quickly changing volume of urban data streams (scalability).

Demand-Centric Integration Methods

Demand-centric integration usually uses a one-time data integration process based on the use case for a specific task (e.g., energy network modeling). Data visualization is one form of demand-centric data integration and has been applied to temporally and interactively integrate data for specific tasks such as analyzing the spatiotemporal variations of energy-use intensity (Sun et al. 2013). However, major drawbacks of visual data integration are that it can result in information loss (Lins et al. 2013) and can be onerous to operate because it often requires a large amount of trial and error to adequately explore a data set (Gu and Wang 2013). Additionally, previous works (Bocconi et al. 2015; Lopez et al. 2012) have used predefined schemas representing certain task demands to integrate fixed data, sensor data, and live social media data, whereas other studies have aimed to directly integrate the knowledge embedded in urban data through machine learning (Zheng 2015) or computational typological analysis (Doraiswamy et al. 2014). Although they are useful for specific tasks, such methods have limited flexibility and reusability because they introduce task-specific biases. For example, the assumptions made for energy network modeling (e.g., modularity and flow) are incompatible with human–building interaction analysis. If there is any change in scope, purpose, or data availability, the task and/or data integration process must be redesigned and reimplemented, which is both time-consuming and computationally expensive (Sheridan and Tennison 2010). As a result, the current demand-centric data integration methods are not flexible and scalable enough to accommodate changes in assumptions, systems, and data availability that are common in the context of urban systems analysis.

Data-Centric Integration Methods

Data-centric integration methods use computational methods to learn the structure, properties, and representation of urban data and then integrate the data based on such attributes. Organizations such as the Open Geospatial Consortium (OGC) have published open standards such as CityGML (OGC 2016) and the Resource Description Framework (RDF) (Miller 1998) which manage data based on a standardized data model and exchange format. Relational databases with flexible schema (Ziegler and Dittrich 2004) and rasters (Tollefsen et al. 2012) have also been used to integrate typologically and spatially heterogeneous data but are limited in their ability to handle temporally inconsistent data that are common in urban environments (Kitchin 2014; Wiemann and Bernard 2016). Similarly, automated systems have been developed to convert tabular data to graphic representations or object-oriented models by identifying and analyzing the structure, content, and semantic attributes of databases (Han et al. 2008; Venetis et al. 2011). However, such methods are again limited in their ability to integrate temporally heterogeneous data (Servigne et al. 2016). Perhaps most importantly, data-centric integration methods are not self-explanatory, and therefore lack interpretability (Castellani Ribeiro et al. 2015). As a result, data-centric integration methods make it difficult to properly formulate queries for data retrieval without extensive expert knowledge (Ferreira et al. 2013) even when new advanced querying methods are utilized (Aguilera et al. 2016). As a result, such methods may have limited applicability in the city context because municipal officials, policy makers, and engineers are unlikely to base key urban design and operational decisions on results and analytical queries that they cannot understand.

Koperski and Han (1995) proposed a generic algorithm to derive rules of association by querying on properties of available urban data and combining properties to increasingly refine results. The approach prefilters elements based on task-specific nonspatial properties and the approach does not define how it would be applied to integrate various types of urban elements with different shapes (e.g., polygon, line, and point) and their data streams. The overall objective of the present paper is to introduce an urban data integration framework for integrating heterogeneous urban data that extends the generic approach suggested by Koperski and Han (1995). The UDI framework is designed to be extensible to multiple types of urban systems, scalable to the quickly changing and growing urban data streams, and interpretable enough to inform municipal decision-making. The UDI framework proposed in this paper focuses on the management of urban data, including filtering, reconstructing, linking, and storing data from different urban systems. The underlying goal of this work is to provide an unbiased basis for enabling specific analytical tasks emerging for the smart city domain such as system prediction and fault detection (Zheng et al. 2015), spatial-temporal analysis (Van Hove et al. 2015), and cross-system impact forecasting (Lund et al. 2015).

UDI Framework

The proposed UDI framework uses a series of novel proximity relationship learning algorithms to automatically integrate urban data in a graph database. In turn, the graph database can then be efficiently explored and queried to answer specific questions regarding urban systems. For example, a decision maker can ask the question "How does a building's proximity to trees and roads impact its energy usage?" The UDI framework is composed of input, three analytical steps, and output (Fig. 1):

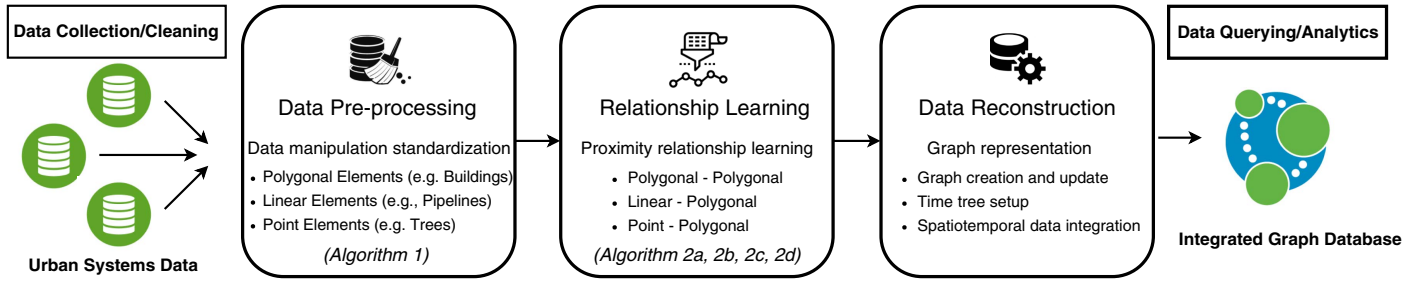

Fig. 1. Overview of UDI framework.

Table 2. Required and optional fields of three categories of urban elements

Category	Required fields	Optional fields
Polygon elements	Geometric coordinates of vertexes of polygon Unique identification number (UID) as primary key	Element-specific properties
Linear elements	Line: geometric coordinates of end points Curve: geometric coordinates of end points for approximated small straight-line segments Unique identification number (UID) as primary key	Element-specific properties
Point elements	Geometric coordinates of point Unique identification number (UID) as primary key	Element-specific properties

Input: Urban data streams (e.g., time-series energy-usage data, building footprints, road outlines, and tree locations).

Analytical Steps

1. Preprocess urban data streams in order to standardize data manipulation for different types of urban elements: polygonal (e.g., buildings and parking lots), linear (e.g., roads and pipelines), and point (e.g., trees and traffic lights) elements (Algorithm 1).
2. Learn the proximity relationships between elements to link disparate urban systems without introducing task-specific bias or assumptions (Algorithms 2a, 2b, 2c, and 2d).
3. Reconstruct the data in a graphic database to enable easy query formulation and efficient data retrieval.

Output: Integrated graphic database on which analytical queries can be easily executed

Data Preprocessing (Step 1)

After various urban data streams are collected and cleaned to remove inconsistencies and errors (e.g., missing geometric information or numeric values for categorical attributes), a three-step preprocessing step is conducted: (1) preparing and reformulating geometric information; (2) converting geometric coordinates into Cartesian coordinates; and (3) identifying the sides of elements that cannot be abstracted as points, such as buildings and roads.

Prepare and Reformulate Geometric Information

The first step is to translate various urban data streams from different sources into consistent formats and reformulate the geometric information of elements for all urban systems. If the information for a required field is missing, the record is removed. There could be any number of optional fields for element properties (Table 2). For example, in addition to geometric information, buildings (polygon elements) could have properties related to height, number of floors, use type, and building system. Similarly, roads (line elements) could have properties of length, width, traffic control, and maintenance schedules; and trees (point elements) could have properties of age, type, height, and shape.

Convert Geometric Coordinates to Cartesian Coordinates

The second step is to convert the geometric coordinates [latitude (α) and longitude (β)] of all elements to Cartesian coordinates (x, y) in the directions $\hat{X}$ and $\hat{Y}$. The equations of conversion between geometric coordinates and Cartesian coordinates are based on the Gall Stereographical Process (Snyder 1987). The Python package PyProj was used to convert the coordinates between the geometric and Cartesian systems. The package uses the following equations:

$$x = \beta/\sqrt{2} \tag{1}$$

$$y = R \times \left(1 + 1/\sqrt{2}\right) \times \tan\left(\alpha/2\right) \tag{2}$$

$$\alpha = 2\tan^{-1}[y/(1 + 1/\sqrt{2})] \tag{3}$$

$$\beta = \sqrt{2}x \tag{4}$$

Identify the Sides of Nonpoint Elements

The third step is to identify the sides of the polygon and linear elements. This step provides the basis to consistently and precisely determine the relative positions of nonpoint elements and the representations of their proximity relationships.

Polygon Elements. Polygon elements have definite shapes and multiple vertices in the $x - y$ plane from the top view. An algorithm (Algorithm 1) to identify the sides of polygon elements was implemented. This algorithm uses the well-established computational geometry method approach suggested by De Berg et al. (2000) and applies it to the context which does not require triangulation of the polygonal elements. The pseudocode is as follows:

Input:
Array V containing vertices of the polygon element
Output:
Array O containing the names of sides of the non-point element and arrangement order of the vertices

$t_x \leftarrow$ x coordinates of V
$t_y \leftarrow$ y coordinates of V
$P \leftarrow t_x$. index min t_x \\ *give the vertex with smallest x coordinate*
$L \leftarrow$ length of array V
$S \leftarrow$ an array of length equal to number of vertices and initialized with value -1 \\ *store the names of sides*
$P^+ \leftarrow (P+1)\%L$ \\ *give the index of vertex after P in the array*
type $\leftarrow$ "clockwise"
if y coordinate $P^+ \geq$ y coordinate of P \\ *indicate clockwise array*
    for c $\leftarrow$ 0 to L-1
        $S[(c+p)\%L] \leftarrow c+1$
else \\ *indicates anti-clockwise array*
    type $\leftarrow$ "anticlockwise"
    for c $\leftarrow$ 0 to L-1
        $S[(c+p)\%L] \leftarrow L-c$
$O \leftarrow$ [S,type]

As an illustration, the implementation of Algorithm 1 on a polygon element A with four vertexes V_1, V_2, V_3, and V_4 is presented in Fig. 2.

The array of Cartesian coordinates for A is given in Table 3. It is assumed that the first element of the array (with index of 0) may be any of the four vertices of A and the order of vertices of A is clockwise (the processing will be same if the vertices of the array are recorded in anticlockwise order).

First, the vertex with the smallest x-coordinate is searched and set as the starting vertex for side identification, which is assumed to be V_4 in the preceding example. Accordingly, the first side in clockwise direction from the starting vertex along the clockwise direction is named Side 1 and the last side is named Side x, where x is the number of sides the element has. Once Side 1 and Side x are identified, other sides can be sequentially determined. In this example, the polygon element A has four sides; V_4 is the starting vertex and the vertex after V_4 in the array is V_1. Because V_{1y} is greater than V_{4y}, the direction from V_4 to V_1 is clockwise, and thus the side between them is named Side 1. The resulting output array O contains the sides of the polygon element and the order of the vertices in the array V.

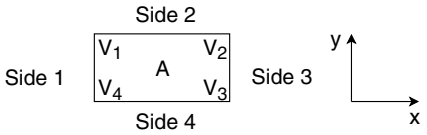

Fig. 2. Example of identifying sides of polygon element.

Table 3. Coordinates of four vertexes of polygon element A

Array element index	Array element geocoordinates (V)
0	$[V_{4x}, V_{4y}]$
1	$[V_{1x}, V_{1y}]$
2	$[V_{2x}, V_{2y}]$
3	$[V_{3x}, V_{3y}]$

Algorithm 1 is generalizable to polygon elements with different number of sides. However, some sides may essentially be the same in terms of representing proximity relationships with other elements, and identifying all of them separately could result in unnecessary redundancy, complexity, and confusion in further relationship learning. For example, Fig. 3(a) shows a building with 10 vertexes and 10 identified sides, but some sides, such as Side 1, Side 3, and Side 9, can be clustered because they represent the same face of the building for defining the proximity relationships with other elements. Therefore, the minimum bounding shaping algorithm (O'Rourke 1985) is applied to approximate a complicated polygon element with simplified side names. The 10-sided building can be represented by four clustered sides [Fig. 3(b)].

This approximation method is applicable to polygon elements of different shapes. For example, the approximation results for basic building shapes, including L, H, U, and T, based on architectural logic/shape grammar (Mitchell 1990) are shown in Fig. 4. The side approximation is only applied to identify the sides of the polygon elements and is not used for further calculating proximity relationship among elements.

Linear Elements. Linear elements are elements that can be represented (straight-line elements) as or segmented (curve elements) into straight lines, each of which have two end points. Following the methods for finding point locations (De Berg et al. 2000), we implement the following approach for finding the side of the linear element (or the segmented linear element), on which a point lies. The coordinates of the end points of the linear element are first used to formulate the equation of the element in the $x-y$ plane. Assuming the coordinates of end points of the element to be $\mathbf{p_1} = (a_1, b_1)$ and $\mathbf{p_2} = (a_2, b_2)$, the equation of the element is the represented as

$$ax + by + c = 0 \tag{5}$$

where $a = (b_2 - b_1)$, $b = (a_1 - a_2)$, and $c = (b_1 \times b_2 - a_1 \times a_2)$. Any point (x_1, y_1) on the $x-y$ plane then has three possibilities: on

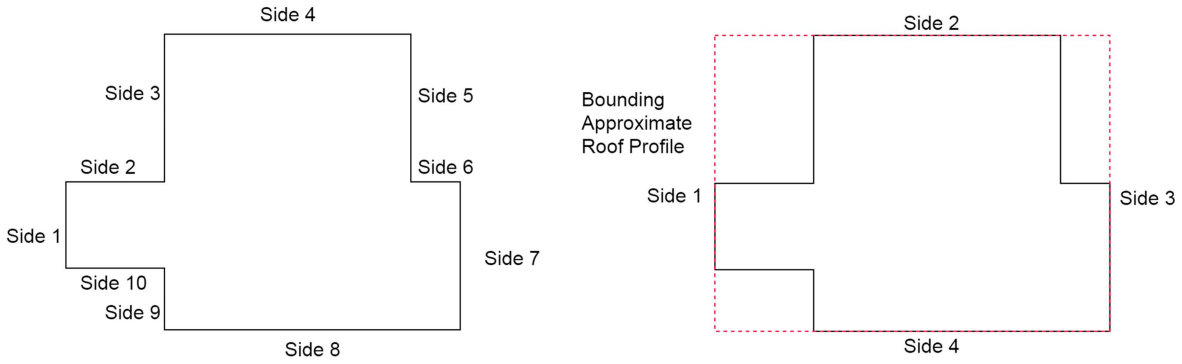

Fig. 3. (a) Building with 10 sides; and (b) building with 4 clustered sides.

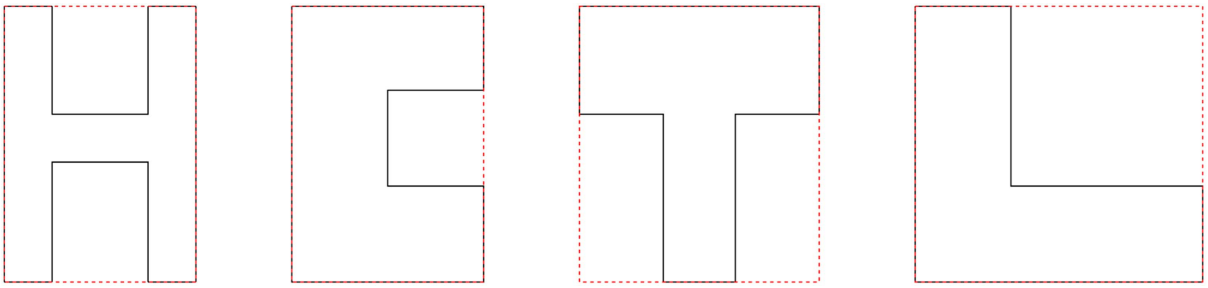

Fig. 4. Approximation results for basic building element shapes.

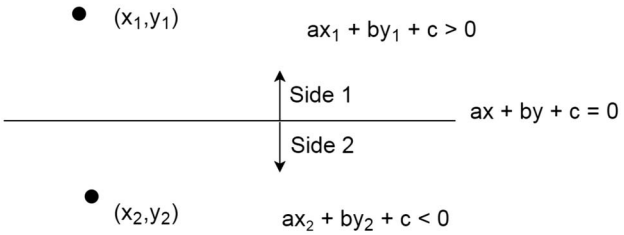

Fig. 5. Example of identifying sides of linear element.

Side 1 of the linear element if $ax_1 + by_1 + c > 0$, on Side 2 of the linear element if $ax_1 + by_1 + c < 0$, and exactly on the linear element if $ax_1 + by_1 + c = 0$. An example is shown in Fig. 5 to illustrate this process.

Proximity Relationship Learning (Step 2)

After urban data streams are uniformly processed, the next step is to learn the proximity relationships of elements, which are unbiased connections for elements, under the following four assumptions:

1. Polygon and linear elements cannot be abstracted to single points and might have irregular shapes; therefore proximity relationships with other elements should be represented by the proximities of other elements to certain sides of polygon and linear elements.
2. A proximity relationship exists between two elements if the shortest distance between the two elements being considered is within a threshold predefined by municipal officials, which can vary for different categories of urban elements. For example, the threshold for proximity relationships of polygon–polygon (e.g., building–building) elements might be 20 m, the threshold for proximity relationships of polygon–point (e.g., building–tree) elements might be 10 m, and the threshold for proximity relationships of line–point (e.g., road–tree) elements might be 30 m, depending on the actual urban context.
3. Because all elements do not have addresses and an address is not informative enough to delineate the proximity relationships with other elements, this framework does not use any address information for learning the proximity relationships in order to maintain generalizability.
4. Proximity relationships can exist between elements of the same urban system (i.e., intraclass proximity relationships such as buildings and buildings) as well as between elements of different urban system (i.e., interclass spatial relationships such as buildings and roads).

The proposed proximity relationship learning process starts by iteratively examining and filtering the elements that have a possibility of being relevant to another element. In order to reduce the computational complexity, the search space of geometric coordinates is limited to a rectangle formed around the target element bounded by the following distances: 200 m to the south of the element's lowest latitude, 200 m to the north of its highest latitude, 200 m to the east of its lowest longitude; and 200 m to the west of its highest longitude. The search space could be changed by municipal officials in order to contextualize the method to their specific cities.

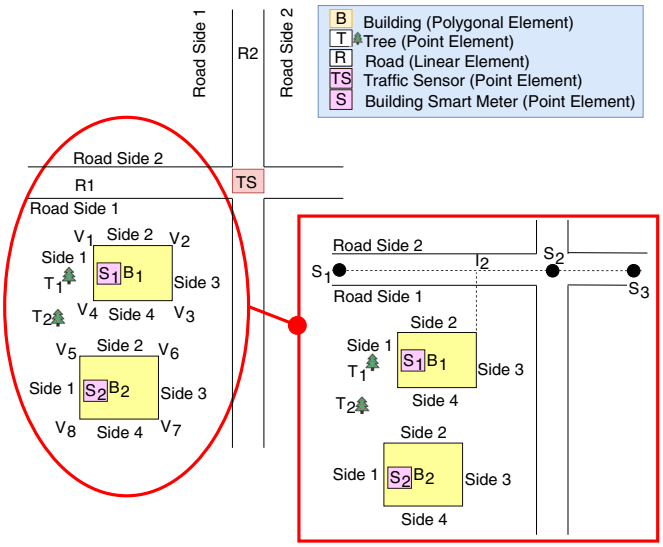

Fig. 6. Example urban parcel for illustrating learning of proximity relationships (Algorithms 2a, 2b, 2c, and 2d).

Next, the framework aims to learn the proximity relationships of elements within the search space. In order to expressively explain the proposed algorithms for this task (Algorithms 2a, 2b, 2c, and 2d), an example of an urban parcel is used (Fig. 6). A polygon element B_1 is located at the corner of the intersection of two linear elements R_1 and R_2. A polygon element B_2 is located along the linear element R_2. A point element T_1 is located on Side 1 of polygon element B_1 and another point element T_2 is located near polygon element B_2. The vertexes of polygon elements B_1 and B_2 were numbered after preprocessing; S_1 and S_2 represent point elements generating time series data and are located inside polygon elements B_1 and B_2, respectively; and TS is a point element generating time series data (i.e., a traffic sensor) but located along linear elements R_1 and R_2. Algorithms 2a, 2b, 2c, and 2d aim to learn the proximity relationships (both interclass relationships and intraclass relationships) for these elements.

Polygon Element and Polygon Element

Algorithm 2a is designed to learn proximity relationships between polygon elements, such as buildings, ponds, fields, parking lots, and so on. The pseudocode for Algorithm 2a is as follows:

Input:
Array A containing vertices of polygon 1
Array B containing vertices of polygon 2
Output:
Boolean R - value indicating if A is proximate to B
Int S showing the side of A to which B is proximate to; initialized to -1

$R \leftarrow$ false
$S \leftarrow$ -1
$A_{appx} \leftarrow$ approximated profile of the vertices of A
$B_{appx} \leftarrow$ approximated profile of the vertices of B
$S_A \leftarrow$ Sides of A_{appx} and order (anticlockwise/clockwise) of vertices
$[N_{AB}, D_{AB}] \leftarrow$ Nearest Neighbors (A,B) \\ *returns a matrix* (N_{AB}) *showing vertices of* B *closest* to \\ *vertices of A and the corresponding distance* (D_{AB})
$d \leftarrow \min(D_{AB})$ \\ d *is the minimum distance between* A *and* B
$a \leftarrow D_{AB}.\text{index}(d)$. \\ a *is the index of the vertex in Array* A *which is closest to* B
$b \leftarrow N_{AB}[a]$ \\ b *is the index of the vertex in Array* B *which is closest to* A

$[N_{AB_{appx}}, D_{AB_{appx}}] \leftarrow \text{NearestNeighbors}(A_{appx}, B_{appx})$
$d_{appx} \leftarrow \min(D_{AB_{appx}})$
$a_{appx} \leftarrow D_{AB_{appx}}.\text{index}(d_{appx})$

\\ *AdjacentVertex: finds indexes of vertices adjacent to vertex with index a in Array A*
$[a^+, a^-] \leftarrow \text{AdjacentVertex}(A, a)$
$[b^+, b^-] \leftarrow \text{AdjacentVertex}(B, b)$
$[a^+_{appx}, a^-_{appx}] \leftarrow \text{AdjacentVertex}(A_{appx}, a_{appx})$

\\ *DistanceBetweenLines: finds the shortest distance be line segments given by the end points given in the arguments*
$d_1 \leftarrow$ Distance Between Lines$(A[a^-], A[a], B[b^-], B[b])$
$d_2 \leftarrow$ Distance Between Lines$(A[a], A[a^+], B[b^-], B[b])$
$d_3 \leftarrow$ Distance Between Lines$(A[a], A[a^+], B[b], B[b^+])$
$d_4 \leftarrow$ Distance Between Lines$(A[a^-], A[a], B[b], B[b^+])$

$d \leftarrow \min(d_1, d_2, d_3, d_4)$
if $d >$ THRESHOLD
    $R \leftarrow$ false
    return
$R \leftarrow$ true

\\ *FindOverlap: finds projection of polygon B on line segment given by vertices with indices* a^- *and* a
$O_1 \leftarrow \text{FindOve lap}(B, A_{appx}[a^-_{appx}], A_{appx}[a_{appx}])$
$O_2 \leftarrow \text{FindOverlap}(B, A_{appx}[A_{appx}], A_{appx}[a^+_{appx}])$

\\ *ExtractSide: finds the side of the array S_A having vertices with indices* a^-_{appx} *and* a_{appx}
if $O_1 > O_2$
    $S \leftarrow \text{ExtractSide } S_A, A_{appx}[a^-_{appx}], A_{appx}[a_{appx}]$
else if $O_2 > O_1$
    $S \leftarrow \text{ExtractSide } (S_A, A_{appx}[a_{appx}], A_{appx}[a^+_{appx}])$
else
    return

Algorithm 2a starts by identifying the vertexes (a, b) of the two polygon elements which are the closest to each other. Then the sides adjacent to these two vertexes are used to calculate the distance between the two polygon elements and identify the side of one polygon element to which the other polygon element is proximate. If the distance between the two polygon elements is less than the threshold, the algorithm proceeds forward and finds the appropriate sides; otherwise it returns the default values (false, −1). The side of polygon element A to which polygon element B is proximate is determined by projecting the polygon element B on the two adjacent sides of A given by ($A_{appx}[a^-_{appx}]$, $A_{appx}[a_{appx}]$) and ($A_{appx}[a_{appx}], A_{appx}[a^+_{appx}]$) and by selecting the side that has a larger projection of B. This is because the projection of B on a side of A is a proxy for calculating the portion of B which faces that side of A. If perpendiculars are drawn from the vertices of end points of the side of A toward B, a higher projection would mean that a larger portion of building B lies between the two perpendiculars. The polygon elements A and B here are interchangeable and thus the same algorithm would also give the side of B to which A is proximate. Specifically, the FindOverlap function to calculate projection of polygon element on one side of another polygon element in the pseudocode works as follows.

The vector equation of the side using the vertexes is denoted as

$$\mathbf{r} = \mathbf{a} + t(\mathbf{b} - \mathbf{a}) = \mathbf{a} + t\mathbf{v} \tag{6}$$

where $\mathbf{v} = \mathbf{b} - \mathbf{a}$, where $\mathbf{a}$ and $\mathbf{b}$ are the vertexes of the side (given as vectors); and t = scaling parameter. The equation of a linear element which is perpendicular to Side ab and passes through a vertex (e.g., p_1) of the polygon element can be expressed as

$$\mathbf{r_1} = \mathbf{p_1} + t_1\mathbf{v}' \tag{7}$$

where t_1 = scaling parameter and $\mathbf{v}'$ = vector perpendicular to $\mathbf{v}$. The point of intersection of $\mathbf{r}$ and $\mathbf{r_1}$ is found, which gives a unique value of t. If $0 < t < 1$, the vertex or intersection lies on Side ab. Therefore, the projection of vertex $\mathbf{p_1}$ on Side ab is given by $i_1 = t$. Similarly, for other vertexes $\mathbf{p_2}, \mathbf{p_3}, \ldots, \mathbf{p_n}$ of the polygon element, the projections $i_2, i_3, \ldots, i_n$, could be calculated. Let $\max_p = \max(i_2, i_3, \ldots, i_n)$ and $\min_p = \min(i_2, i_3, \ldots, i_n)$; the projection of the polygon element on Side ab is defined as intersection of $[\min_p, \max_p]$ with [0, 1] and is given by ordered interval I.

Fig. 6 shows the sides identified for polygon elements B_1 and B_2 for the urban context. In this example, the pair (a, b) as per Algorithm 2a is (V_4, V_5). The two vertexes adjacent to V_4 are V_1 and V_3 and the two vertices adjacent to V_5 are V_6 and V_8. Therefore, the polygon element B_2 can be either on Side 4 or Side 1 of polygon element B_1 and polygon element B_1 can be either on Side 1 or Side 2 of polygon element B_2. If the minimum distance between these two pairs of lines segments is less than the user-defined threshold, the projection of the polygon element B_2 on Side 4 of polygon element B_1 is less than 1. Similarly, the projection of polygon element B_2 on Side 1 of polygon element B_1 is 0. Therefore, it is concluded that the polygon element B_2 is proximate to the Side 4 of polygon element B_1 and polygon element B_1 is proximate to Side 2 of polygon element B_2.

Linear Element and Polygon Element

Algorithm 2b is designed to learn proximity relationships between linear elements (such as roads, pipelines, and rivers) and polygon

elements. If the linear elements are not straight, the segmented straight lines are used instead. The pseudocode for Algorithm 2b is

Input:
Array S containing endpoints of segments of the linear element
Array A containing vertices of the polygon element
Output:
Boolean R showing whether a segment of S is proximate to A
Array P_s showing the end points of segment of S to which A is proximate

Shortest distance d between S and A
Side R_A of A on which S is located
Side R_S of S on which A is located

$R \leftarrow$ false
$C_A \leftarrow$ centroid of the vertices of A
$d_{min} \leftarrow 100$
$s_i \leftarrow 0$
\\ *Following loop finds the segments of linear element which are closest to A*
for $i \leftarrow 0$ to length(S) $-$ 1
    $t \leftarrow$ distance(C_A, S[i])
    if $t < d_{min}$
        $d_{min} \leftarrow t$
        $s_i \leftarrow i$
if $S_i =$ length(S) $-$ 1
    $S_r \leftarrow [S[s_i - 1], S[s_i]]$
else if $S_i = 0$
    $S_r \leftarrow [S[s_i], S[s_i + 1]]$
else
    $S_r \leftarrow [S[s_i - 1], S[s_i], S[s_i + 1]]$

$A_{appx} \leftarrow$approximate profile of the vertices of A
S_A Sides of A_{appx} and order (anticlockwise/clockwise) of vertices
$[N_{AS}, D_{AS}] \leftarrow$ NearestNeighbors(A, S)\\ *return a matrix showing point in S closest to every vertex of A and the corresponding distance*
$d \leftarrow \min(D_{AS})$
$a \leftarrow D_{AS}$.index(d)
$s \leftarrow N_{AS}[a]$
$[N_{AS_{appx}}, D_{AS_{appx}}] \leftarrow$ NearestNeighbors(A_{appx}, S)
$d_{appx} \leftarrow \min(D_{AS_{appx}})$
$a_{appx} \leftarrow D_{AS_{appx}}$.index($d_{appx}$)
$[a^+, a^-] \leftarrow$ AdjacentVertex(A, a)
$[a^+_{appx}, a^-_{appx}] \leftarrow$ AdjacentVertex(A_{appx}, a_{appx})

if length(S_r) $= 2$
    $[s^-, s] \leftarrow [0, 1]$
    $d_1 \leftarrow$ Distance Between Lines($A[a^-]$, A[a], $S[s^-]$, S[s])
    $d_2 \leftarrow$ Distance Between Lines(A[a], $A[a^+]$, $S[s^-]$, S[s])
    $d \leftarrow \min(d_1, d_2)$
    if $d > THRESHOLD$
        return
    $R \leftarrow$ true
    $O_1 \leftarrow$ FindOverlap(S_r, $A_{appx}[a^-_{appx}]$, $A_{appx}[a_{appx}]$)
    $O_2 \leftarrow$ FindOverlap(S_r, $A_{appx}[a_{appx}]$, $A_{appx}[a^+_{appx}]$)
    if $O_1 > O_2$
        $R_A \leftarrow$ ExtractSide(S_A, $A_{appx}[a^-_{appx}]$, $A_{appx}[a_{appx}]$)
    else if $O_2 > O_1$
        $R_A \leftarrow$ ExtractSide(S_A, $A_{appx}[a_{appx}]$, $A_{appx}[a^+_{appx}]$)
    else
        return
else
    $[s^+, s^-] \leftarrow$ AdjacentVertex(S, s)
    $d_1 \leftarrow$ Distance Between Lines($A[a^-]$, A[a], $S[s^-]$, S[s])
    $d_2 \leftarrow$ Distance Between Lines(A[a], $A[a^+]$, $S[s^-]$, S[s])
    $d_3 \leftarrow$ Distance Between Lines(A[a], $A[a^+]$, S[s], $S[s^+]$)
    $d_4 \leftarrow$ Distance Between Lines($A[a^-]$, A[a], S[s], $S[s^+]$)
    $d \leftarrow \min(d_1, d_2, d_3, d_4)$
    $d_i \leftarrow [d_1, d_2, d_3, d_4].index(d)$
    if $d_i = 0$ or $d_i = 1$
        $P_s \leftarrow [S[s^-], S[s]]$
    else
        $P_s \leftarrow [S[s], S[s^+]]$
    if $d > THRESHOLD$
        return
    $R \leftarrow$ true
    $O_1 \leftarrow$ FindOverlap(S_r, $A_{appx}[a^-_{appx}]$, $A_{appx}[a_{appx}]$)
    $O_2 \leftarrow$ FindOverlap(S_r, $A_{appx}[a_{appx}]$, $A_{appx}[a^+_{appx}]$)
    if $O_1 > O_2$
        $R_A \leftarrow$ ExtractSide(S_A, $A_{appx}[a^-_{appx}]$, $A_{appx}[a_{appx}]$)
    else if $O_2 > O_1$
        $R_A \leftarrow$ ExtractSide(S_A, $A_{appx}[a_{appx}]$, $A_{appx}[a^+_{appx}]$)
    else
        return
G(x,y) $\leftarrow$ equation of line representing s^- and s (Eq. 5)
if $G(C_{A_x}, C_{A_y}) > 0$
    $R_S \leftarrow 1$
else if $G(C_{A_x}, C_{A_y}) < 0$
    $R_S \leftarrow 2$

The coordinates of segmented lines are represented as an array which contains the end points of the segment in order. Algorithm 2b starts by identifying the end points in the array which are closest to the polygon element. This serves as a preliminary test to select two segments of the linear element which could be closest to the polygon element. After the two segments have been identified, the algorithm finds the vertex of the polygon element which is closest to the identified segments. The shortest distance between the segments and the two adjacent sides of the identified vertex is calculated, and if this distance is more than the threshold, the algorithm ends. If the distance is less than the threshold, the projection of the two segments of the linear element on the two adjacent sides of the polygon (to the vertex identified previously) is calculated. The side of the polygon which has the larger projection is then defined to be proximate to the corresponding segment of the linear segment. Using the center of the polygon, the position of the polygon element with respect to the segment of the linear element is also identified. With respect to the example being discussed, the linear element R_1 consists of two segments with three vertices (Table 4).

For the urban context example in Fig. 6, it is clear that the segment between S_1-S_2 has the shortest distance to the center of polygon element B_1 (vertex V_2 being the vertex closest to S_1-S_2); I_2 is the point of intersection of perpendicular from V_2 on R_1; and the projection of S_1-S_2 on Side 2 is greater than its projection on Side 1. Therefore, segment S_1-S_2 of the linear element R_1 is proximate to Side 2 of the polygon element B_1, and the distance between I_2 and V_2 represents the distance between the polygon element B_1 and the linear element R_1 (Fig. 6).

Table 4. Array of coordinates for one linear element segmented into two straight lines

End point name	Array value
S_1	$[S_{1x}, S_{1y}]$
S_2	$[S_{2x}, S_{2y}]$
S_3	$[S_{3x}, S_{3y}]$

Point Element and Polygon Element

Algorithm 2c is designed to learn proximity relationships between point elements (e.g., trees, sensors, and traffic lights) and polygon elements. The pseudocode for Algorithm 2c is

Input:
Vector P containing coordinates of the point element
Array A containing vertices of the polygon element
Output:
Boolean R showing whether P and A are proximate
Shortest distance d between P and A
Side R_A of A on which P is located

$d \leftarrow 0$
$A_{appx} \leftarrow$ approximate profile of the vertices of A
$S_A \leftarrow$ Sides of A_{appx} and order (anticlockwise/clockwise) of vertices

for $i \leftarrow 0$ to length(A) $-$ 1
    $t \leftarrow$ distance(P, A)
    if $t < d_{min}$
        $d_{min} \leftarrow t$
        $a \leftarrow i$
$[a^+, a^-] \leftarrow$ AdjacentVertex(A, a)

for $i \leftarrow 0$ to length(A_{appx}) $-$ 1
    $t \leftarrow$ distance(P, A_{appx})
    if $t < d_{min}$
        $d_{min} \leftarrow t$
        $a_{appx} \leftarrow i$
$[a^+_{appx}, a^-_{appx}] \leftarrow$ AdjacentVertex(A_{appx}, a_{appx})
$[a^+_{appx}, a^-_{appx}] \leftarrow$ AdjacentVertex(A_{appx}, a_{appx})
$R_A \leftarrow$ ExtractSide($S_A, A_{appx}[a^-_{appx}], A_{appx}[a_{appx}]$)

\\ *DistancePointLine: gives the shortest distance between a line segment and a point P*
$d_1 \leftarrow$ DistancePointLine(P, A[a^-], A[a])
$d_2 \leftarrow$ DistancePointLine(P, A[a], A[a^+])

$d \leftarrow \min(d_1, d_2)$
if $d >$ *THRESHOLD*
    return
\\ *ProjectPointLine: give an integer showing the location of projection of a point P on a line segment given by two end points*
$R \leftarrow$ true
$p_1 \leftarrow$ ProjectPointLine(P, $A_{appx}[a^-_{appx}], A_{appx}[a_{appx}]$)
$p_2 \leftarrow$ ProjectPointLine(P, $A_{appx}[a_{appx}], A_{appx}[a^+_{appx}]$)
if $0 < p_1 < 1$
    $R_A \leftarrow$ ExtractSide($S_A, A_{appx}[a^-_{appx}], A_{appx}[a_{appx}]$)
else if $0 < p_2 < 1$
    $R_A \leftarrow$ ExtractSide($S_A, A_{appx}[a_{appx}], A_{appx}[a^+_{appx}]$)
else
    $R \leftarrow$ false
    $R_A \leftarrow -1$ *The value -1 denotes that the point is proximate to both the sides*

Algorithm 2c first identifies the vertex of the polygon element which is closest to the point element. The adjacent two sides are then used to calculate the shortest distance between the point element and the polygon element. The projections of the point on both sides are calculated and the side for which projection is between 0 and 1 is selected as the side of A that is proximate to the point element. If the projection is outside this range for both sides, then the point element is proximate to both sides (adjacent to the closest vertex) of the polygon element. In Fig. 6, the point element T_1 is proximate to Side 1 of the polygon element B_1 and the point element T_2 is proximate to both Side 1 and Side 2 of the polygon element B_2.

Linear Element to Linear Element

Algorithm 2d is designed to learn proximity relationships between linear elements. The pseudocode for the Algorithm 2d is

Input:
Array L_1 showing points of segments of a linear element
Array L_2 showing points of segments of second linear element

Output:
Array S containing segments of L_2 which are closest to segments of L_1(only if the distance is within the user defined threshold)
Array L_s containing side of L_1 on which closest segments of L_2 are located

for $i \leftarrow 0$ *to length*$(L_1) - 1$
  $d \leftarrow 0$
  for $j \leftarrow 0$ *to length*$(L_2) - 1$
    $t \leftarrow$ *Distance Between Lines*$(L_1[i], L_2[j])$
    if $t < d$
    $d \leftarrow t$
  if $d >$ *THRESHOLD*
    return
  else if $d = 0$
    $L_s[i] = 0$ \\ *line segments intersect*
  else
    $S[i] \leftarrow L_2[j]$
  \\ *find the side of* L_s *on which A is located*
  G(x, y) $\leftarrow$ equation of line representing segment
  $L_1[i][0]$ and $L_1[i][1]$(Eq. 5)
  if $G(L_2[j][0]_x, L_2[j][0]_y) > 0$
    $L_{s1}[i] \leftarrow 1$
  else if $G(L_2[j][0]_x, L_2[j][0]_y) < 0$
    $L_{s1}[i] \leftarrow 2$

For every segment of linear element L_1, the algorithm searches all the segments of linear element L_2 and determines the one that is closest to that segment of L_1. If the distance between them is within the predefined threshold, the two segments are proximate to each other.

Point Element to Linear/Point Element

The learning of proximity relationships between point elements and linear elements is similar to how the DistancePointLine and ProjectPointLine functions in section "Point Element and Polygon Element" work. For the urban context example in Fig. 6, it can be calculated that the point element T_1 is proximate to Side 1 of linear element R_1. The learning of proximity relationships between point elements is done by simply calculating the Euclidean distances using the point coordinates.

Data Reconstruction in Graph (Step 3)

A graph-based structure composed of nodes and directed edges is proposed. Each node represents an element and has attributes to represent properties (e.g., age, height, and area) and time-series data (e.g., electricity usage each hour). Each directed edge has attributes such as numeric distance to represent proximity relationship between elements. Fig. 7 illustrates the graph representation of eight elements comprising four classes (building, road, tree, and sensor) for the example in Fig. 6. In Fig. 7, B_1 and B_2 are two nodes connected by directed edges. Their proximity relationship

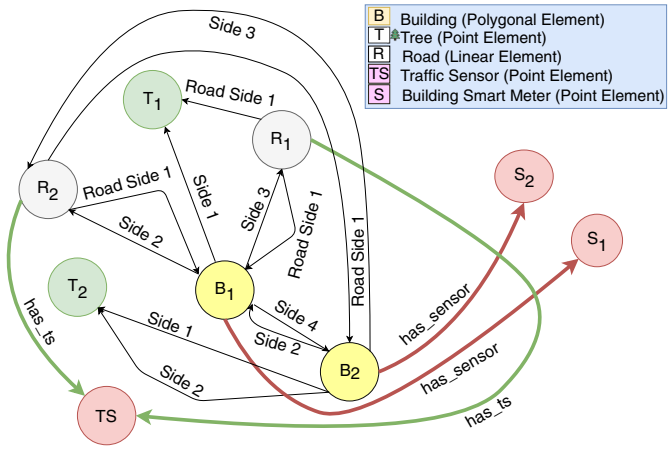

Fig. 7. Graph representation for urban context example (Fig. 6) and their proximity relationships.

can be delineated as polygon element B_1 is proximate to Side 4 of polygon element B_2 of the same system and polygon element B_2 is proximate to Side 2 of polygon element B_1. Both B_1 and B_2 are proximate to linear element R_1 and are on the same side of it. The point element T_1 is proximate to B_1, whereas T_2 is proximate to two sides of B_2. The graph representation of all the elements in this node-edge combination clearly shows how the elements are proximate to each other and how the graph integrates them without bringing in any task specific bias or assumptions. As a result, the graph representation enables a consistent basis for understanding and querying data for specific urban analytical applications.

Particularly, some elements (e.g., sensors) might have both static (e.g., precision and manufacturer) and dynamic (e.g., time-series sensing data) properties with different time scales (e.g., daily and monthly). In order to deal with such elements, the proposed UDI framework includes a hierarchical time structure to store urban data with different time scales ranging from year to second. For example, there are two point elements (e.g., sensors) with measurements of different time scales in Table 5.

The time tree for these two point elements is shown in Fig. 8. All the possible time scales are represented by a hierarchical tree

Table 5. Data with different time granularities from two types of sensors

Point element	Time (yyyy-mm-dd HH-MM)	Value
TS (traffic sensor)	2017-02-13 12-09	74°F
Granularity = Minute	2017-02-13 12-10	80°F
S_1 (building smart meter)	2017-02-13	22 kWh
Granularity = Day	2017-02-14	30 kWh

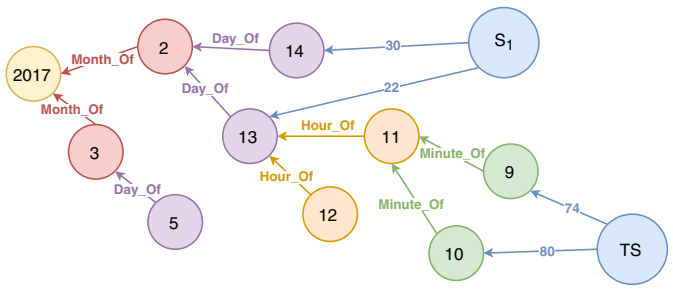

Fig. 8. Time tree constructed to integrate elements with different time scales.

whose nodes can then be connected to elements with measurements as the properties of edges. This time tree–based representation is organized such that various urban data streams with differing time scales (sampling frequencies) can be linked and analyzed. More importantly, when new data become available (e.g., installation of new traffic sensors) or the time scales of existing data sources change (e.g., upgrading of energy meters to smart meters), it is easy to connect them to the existing time tree and continue the process of integrating them with other urban data streams. Further depending on the specific application requirements, the data of different scales could be either converted to other scales or kept with the original scales.

Once all urban data are integrated and stored in the graph representation, the node-edge data structure makes it convenient and efficient to design, formulate, and execute queries for data exploration and retrieval. In this paper, Cypher (Vukotic et al. 2014) is used to complete this process. Cypher is a graph query language designed to be self-explanatory and declarative. It focuses on the clarity of expressing what to retrieve from a graph, and its constructs are based on English prose and compact iconography that in turn make queries more self-explanatory (Miller 2013). For example, in order to include other buildings that are proximate to target Building A for urban heat island analysis, municipal officials only need to list the elements and their relationships using simple expressions such as (A: buildings)-[proximate to]-(all: buildings), return all. Cypher traverses the graph, finds all the matched elements, returns the buildings that are proximate to the target Building A. In addition to data retrieval, reconstructing urban data based on proximity relationships and representing them in graphs make it flexible and expressive to update the existing graph with new data streams and/or elements. The graph is extensible to multiple urban systems and scalable to the variations of data availabilities because the new data can be easily connected with graph structure by proximity relationships. Specifically, to link another element to the existing data, a node needs to be added to the graph with edges determined by the proposed learning framework. Similarly, an element can be removed by deleting the corresponding node on the graph and the edges starting from or ending at that node without affecting how the remaining data functions.

Case Study: Palo Alto, California

In order to illustrate the performance and effectiveness of the UDI framework, a real-world case study was conducted for the city of Palo Alto, a midsize city in the west of the United States. A 1-km^2 study area was selected (Fig. 9) for the implementation of UDI. In this area, data were available for four urban systems: buildings (polygon elements), roads (linear elements), vegetation (point elements), and sensors (point elements with measurements of different time scales). Specifically, sensors in the case study consisted of two types: Type I is Array of Things (AoT)-type sensors (AoT 2018) that monitor the status of the urban environment (including atmospheric pressure, temperature, relative humidity, sound intensity, and vehicle count); Type II is smart meters which are installed in buildings to record the energy consumption, including gas and electricity.

Proximity Relationship Learning

The algorithms for learning the proximity relationships were applied to the data available for this case study area. Details of two building elements are presented in Fig. 10(a) to better illustrate their spatial relationships with other urban elements. Building27509 (27509 is a unique identifier) is proximate to

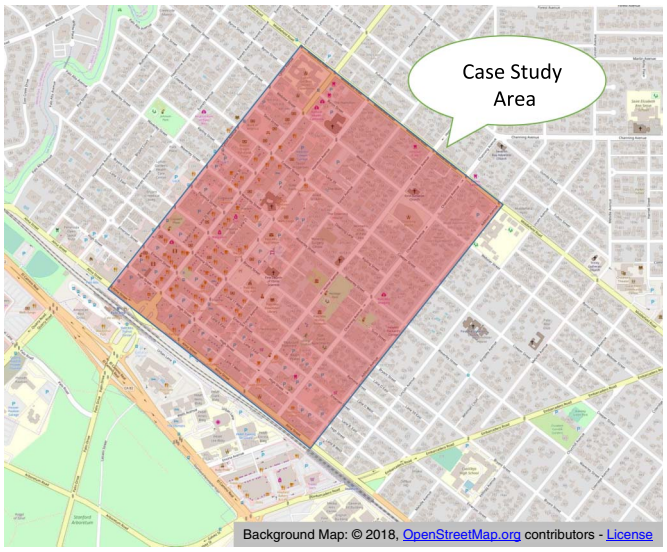

Fig. 9. Case study area randomly selected in city of Palo Alto, California. (Map data © 2018 OpenStreetMap contributors.)

Side_1 of Road117394 and three tree elements. This building is also proximate to three other building elements. Similarly, Building27498 is proximate to Side_2 of Building27509 and has a spatial relationship with the Side_1 of Road 117394. All the information concluded from implementing the proposed algorithms is consistent with the information acquired from a real map visualization [Fig. 10(b)] and thus demonstrates the effectiveness and accuracy of the proximity relationship learning framework.

In addition, the time-tree structure was constructed to integrate data with different time scales. When values are converted between different time scales, there are two types of conversions: (1) conversion of values of measurement from the larger time scale to the smaller time scale; and (2) conversion of values of measurement from the smaller time scale to the larger time scale. In the first case, the values are either divided equally to the smaller time scale from the larger time scale (e.g., electricity consumption) or the same value of the larger time scale is applied to the smaller time scale (e.g., indoor temperature) depending upon the nature of measurement. In the second case, the values from smaller time scale to the larger time scale are calculated by either adding the values of smaller time scale to the larger time scale (e.g., electricity consumption) or using an average (e.g., indoor temperature), which again depends upon the nature of the measurement.

For example, buildings have energy data collected every hour and temperature data collected every second. If the smallest time scale (second) is required for analysis, the hourly building energy data are equally divided by 3,600 to represent the value per second (Table 6), whereas there is no need to convert the temperature values because they were recorded at the seconds time scale. If the minute time scale is the required for analysis, the hourly energy data are equally divided by 60, whereas the temperature data per second are averaged for each minute (Table 7). We used this simple method to reconcile the time-scales and note that evenly distributing total consumption across smaller time scales removes the time-variance

Table 6. Data returned with seconds scale by time tree of UDI framework

Building name	Time (dd-mm-yyyy HH:MM:SS)	Energy value (Wh)	Temperature value (°F)
Building27508	13-02-2017 12:02:01	0.23	73.0
	13-02-2017 12:02:02	0.23	73.2
	13-02-2017 12:02:03	0.23	72.9
	13-02-2017 12:02:04	0.23	73.7
	13-02-2017 12:02:05	0.23	72.4
	13-02-2017 12:02:06	0.23	73.0
	13-02-2017 12:02:07	0.23	73.7
Building27506	14-02-2017 14:45:00	0.28	77.2
	14-02-2017 14:45:01	0.28	76.7
	14-02-2017 14:45:02	0.28	77.8
	14-02-2017 14:45:03	0.28	77.4
	14-02-2017 14:45:04	0.28	76.9
	14-02-2017 14:45:05	0.28	76.2
	14-02-2017 14:45:06	0.28	77.5

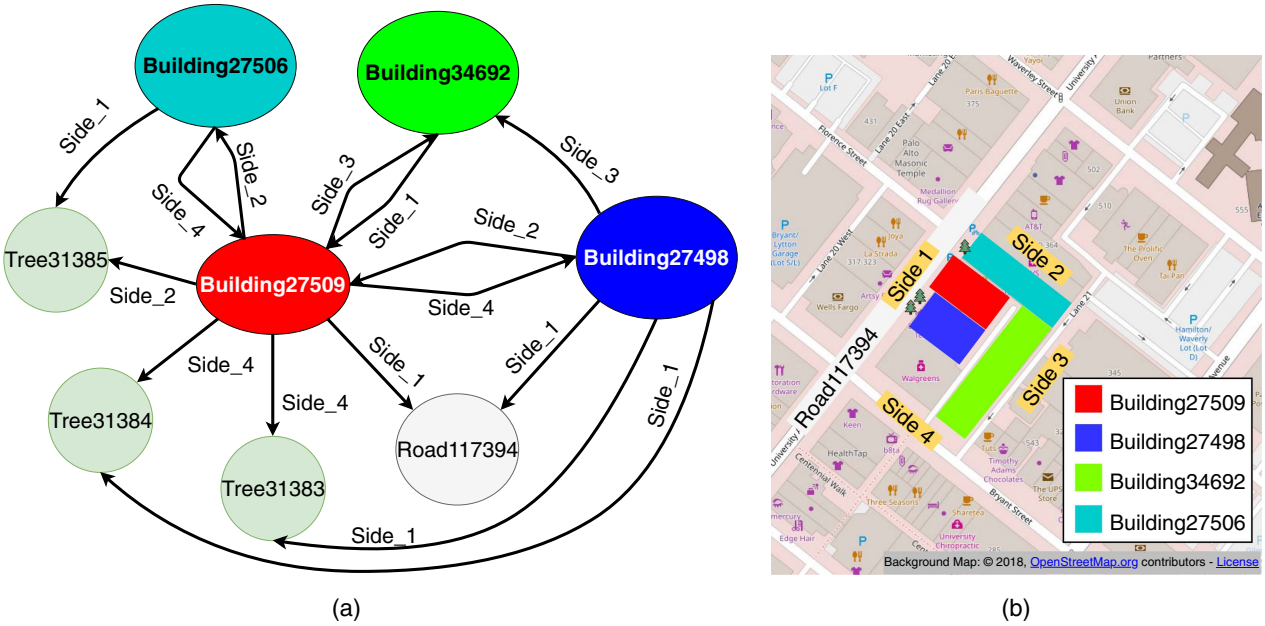

Fig. 10. (a) Details of two building elements and their proximity relationships with other elements; and (b) corresponding locations on actual map visualization (map data © 2018 OpenStreetMap contributors).

Table 7. Data returned with minutes scale by time tree of UDI framework

Building name	Time (dd-mm-yyyy HH:MM)	Energy value (Wh)	Temperature value (°F)
Building27509	13-02-2017 12:02	14	73
	13-02-2017 12:04	14	74
	13-02-2017 12:05	14	75
Building27506	14-02-2017 14:45	17	77
	14-02-2017 14:47	17	77
	14-02-2017 14:48	17	78
	14-02-2017 14:49	17	77

aspects at the lower time scale. This method is meant to be illustrative in nature; users can use domain-specific models if they wish to preserve time-variance aspects and eliminate the assumption of uniform consumption at the lower time scale. For example, users could use a typical energy load curve for the specified building type to distribute total consumption at smaller time scales.

Data Querying and Retrieval

The UDI framework aims to provide an easily interpretable and straightforward way for municipal officials with minimal experience and knowledge of databases to understand and query urban data for further analysis. The graph representation of urban data is intended to simplify the data query design and formulation process and reduce the amount of time investment necessary to conduct analysis. To illustrate the interpretability and computational efficiency of the UDI framework, this paper explores an example of an urban analytics application and benchmarks the performance of the UDI framework against an existing data management method (i.e., a relational database). In this example, it is assumed that municipal officials aim to explore the mutual influences between outside temperature and building energy consumption. They want to eliminate the effects of vehicles passing by the buildings, which could cause disturbances to the thermal environment and temperature. In addition, they want to consistently compare buildings and minimize the confounding effects of shading from trees on energy consumption.

In order to perform this analytical query, the following data should be appropriately retrieved: "energy consumption data of buildings from Type II smart meters and temperature values from Type I temperature sensors proximate to these buildings, on the condition that the buildings have at least four proximate trees of type oak, when the Type I traffic sensors show no vehicles around the buildings." In a traditional data management setting, the data are stored in a relational database as tables (Table 8), whereas in the proposed UDI framework the data are stored in a graph.

First, Structured Query Language (SQL) queries were designed to retrieve the appropriate data for the analytics example using a traditional relational database. Particularly, this query involves creating and executing seven complex joins across multiple tables:

```
SELECT AOT.Time, b_select.building_name INTO time_values
FROM Building_relationships as br2
JOIN (SELECT b.building_name
        FROM Buildings AS b
        JOIN Building_Relationships AS br
            ON b.building_name = br.building_name
            AND br.proximate_object_type="Tree"
        JOIN Trees AS t
            ON t.tree_name=br.proximate_object_name
            AND t.tree_type = "Oak"
        GROUP BY building_name
        HAVING COUNT(*) >= 5) AS b_select
      ON b_select.building_name = br2.building_name
      AND br2.proximate_object_type='AOT'
JOIN AOT_Vehicle_Sensor as AOT
      ON br2.proximate_object_name=AOT.sensor_name
      AND AOT.sensor_value=FALSE

SELECT * INTO br_new
FROM time_values AS tv
JOIN Building_Relationships AS br3
      ON tv.building_name = br3.building_name

SELECT r2.building_name AS bn, MONTH(r2.sensor_time) AS m, YEAR(r2.sensor_time) AS y, DAY(r2.sensor_time) AS d,
HOUR(r2.sensor_time) AS h, MINUTE(r2.sensor_time) AS min, r2.values as val
INTO Energy_Data
FROM (SELECT r1.building_name, r1.sensor_time, Energy.electricity_use INTO r2
      FROM (SELECT * FROM br_new WHERE br_new.proximate_object_type="Energy_sensor") AS r1
        JOIN Smart_Meter AS Energy
            ON r1.proximate_object_name=Energy.sensor_name
            AND MONTH(r1.sensor_time)= MONTH(Energy.sensor_time)
            AND DAY(r1.sensor_time)= DAY(Energy.sensor_time)
            AND YEAR(r1.sensor_time)= YEAR(Energy.sensor_time)
            AND HOUR(r1.sensor_time)=HOUR(Energy.sensor_time))
ORDER BY r2.building_name, YEAR(r2.sensor_time), MONTH(r2.sensor_time), DAY(r2.sensor_time), HOUR(r2.sensor_time),
MINUTE(r2.sensor_time)
```

```
SELECT r4.building_name AS bn, MONTH(r4.sensor_time) AS m, YEAR(r4.sensor_time) AS y, DAY(r4.sensor_time) AS d,
HOUR(r4.sensor_time) AS h, MINUTE(r4.sensor_time) as min, AVG(r4.values) as average
INTO Temperature_Data
FROM (SELECT r3.building_name,Temp.sensor_time, Temp.sensor_value INTO r4
      FROM (SELECT * FROM br_new where br_new.proximate_object_type="Temp_sensor") AS r3
          JOIN AOT_Temperature_Sensor AS Temp
              ON r1.proximate_object_name=Temp.sensor_name
              AND MONTH(r3.sensor_time)= MONTH(Temp.sensor_time)
              AND DAY(r3.sensor_time)= DAY(Temp.sensor_time)
              AND YEAR(r3.sensor_time)= YEAR(Temp.sensor_time)
              AND HOUR(r3.sensor_time)=HOUR(Temp.sensor_time)
              AND MINUTE(r3.sensor_time)=MINUTE(Temp.sensor_time))
GROUP BY r4.building_name, YEAR(r4.sensor_time), MONTH(r4.sensor_time), DAY(r4.sensor_time), HOUR(r4.sensor_time),
MINUTE(r4.sensor_time)
```

As a result,

```
MATCH (s1: AoT {type: "I", name: "Traffic"}) – [p1: ProximateTo] -> (b: Building) <- [p2: ProximateTo] – (s2: AoT {type: "I", name:
"Temperature"})
WHERE p1.value <= threshold AND p2.value <= threshold
WITH s1, s2, p1, p2, b
MATCH (s1) – [v1: Value] -> (m: Minute) – [: Contains] -> (s: Second) <- [v2: Value] – (s2)
WHERE v1= 1
WITH s1, s2, p1, p2, b, m, s, v1, v2
MATCH (s3: Meter {type: "II", name: "SmartMeter"}) – [:LocatedIn] -> (b) <- [p3: ProximateTo] – (t: Tree {type: "Oak"})
WHERE p3.value <= threshold
WITH s1, s2, s3, p1, p2, p3, b, m, s, v1, v2
MATCH (s3) – [v3: Value] -> (h: Hour) – [: Contains] -> (m)
UNWIND COLLECT(h.name)
WITH s1, s2, s3, p1, p2, p3, b, h, m, s, v1, v2, v3
RETURN h.name, v3.value, avg(v2.value)
```

It is clear from this comparison that the design and formulation of SQL queries for relational databases is relatively more complicated and requires expert knowledge and experience to execute. Municipal officials in general do not have the required expertise and/or time to design lengthy queries. As a result, they require more-intuitive means of querying data. Moreover, the SQL queries are not reproducible, which leads to a large amount of unnecessary repeated efforts. In contrast, the design and formulation of queries using Cypher are intuitive because they describe elements and their relationships. Only slight modifications to the queries are required when data availability changes or the metadata schema varies.

The second comparison is with respect to the computational performances between the UDI framework and traditional relational database. The magnitude of time complexity is used as the metric for this comparative analysis. In a relational database, all the elements in the joint tables have to be searched for each target element, which exponentially increases the time complexity. In contrast, in a graph database the proximate elements to be searched are simply the ones connected to the target elements in the graph. As a result, the time complexity of the SQL query is $O[n_c \times n_b \times n_{br} \times n_t \times (n_e + n_{\text{temp}})]$ (Table 8 defines the variables), which can be approximated as $O(n^5)$, and the complexity of executing the Cypher query is $O(n_t) + O(n_c) + O(n_e) + O(n_{\text{temp}})$, which can be approximated as $O(n)$. Appendixes I and II provide details on how the time complexity of the SQL query and the Cypher query were computed. The complexity calculation for the SQL query on relational database management system (RDBMS) would be sensitive to the schema of the SQL database. However, the high time

Table 8. Metadata of tables in relational database

Attribute name	Data type
Building relationships (n_{br} rows)[a]	
building_name	String
proximate_object_type	String
proximate_object_name	String
proximate_object_distance	Float
Smart meter (n_e rows)[b]	
sensor_name	String
Time	YYYY-MM-DD HH
electricity_use	Float
AOT vehicle sensor (n_c rows)[c]	
sensor_name	String
Time	YYYY-MM-DD HH:MM
sensor_value	Binary
AOT temperature sensor (n_{temp} rows)[d]	
sensor_name	String
Time	YYYY-MM-DD HH:MM:SS
sensor_value	Float
Buildings (n_b rows)[e]	
building_name	String
Trees (n_t rows)[f]	
tree_name	String
tree_type	String

[a]Proximity relationships of elements.
[b]Values of smart meters for building energy consumption.
[c]Values of vehicle sensors to monitor whether vehicles are present.
[d]Values of temperature sensors proximate to buildings.
[e]Properties of buildings.
[f]Properties of trees.

complexity of SQL queries would still hold true because it involves the execution of joins between tables, which are expensive. The graph database eliminates the need for expensive joins, and this difference in time complexities in turn gets magnified when the number of nested joins increases in corresponding SQL queries. Therefore, along with the simplified design and formulation of queries, reconstructing urban data based on proximity relationships and storing them in a graph database can enable faster executions of queries and data retrieval compared with data management using a relational database. This is especially important for urban analytics applications, because often the queries are explorative in nature and thus require numerous iterations and combinations to be run.

Limitations and Future Work

Although the proposed UDI framework addresses several gaps in the urban data management literature, it has several limitations that could be addressed in future work. First, the graph representation is designed for individual elements and their relationships because it is not practical and feasible to visualize and analyze all elements in the entire urban area simultaneously. Future work could begin to overcome this limitation by developing clustering methods to classify urban elements into meaningful classes based on the patterns and properties of their proximity relationships. Integration of UDI with some well-established geographic concepts such as geo-atom (Goodchile et al. 2007) could be explored for better representing the hierarchical proximity relationships among urban elements and improving analysis and visualization. This would enable groups of elements to be represented in the graph database structure and facilitate analysis across multiple element groups. A second limitation is that the proposed framework is only able to infer two-dimensional (2D) proximity relationships in urban data, and thus ignores the vertical dimension of a city. Although this is a simplification of the proximity inference problem potentially resulting in false positives, learning the 2D relationship represents a strong first step in the urban data integration process, and future work could extend this work to incorporate vertical context in a three-dimensional (3D) proximity relationship algorithm. As with most data-driven based research, the UDI framework could benefit from further validation. This paper used data from a typical midsize city in the United States (Palo Alto, California) to validate the framework, but future work should extend this validation to larger, more complex urban environments both in the United States (e.g., San Francisco and New York) and internationally (e.g., London, Beijing, Rio de Janeiro, and Mumbai). Additionally, further data could be collected on the usability and interpretability aspects of the proposed framework through surveys and interviews of users. Doing so would enable deeper insights into how users utilize and interpret queries in the UDI framework and would further corroborate the query length/syntax analysis undertaken in this work. Lastly, future work could also address many of the practical challenges of urban data integration and analytics, such as data ownership, control, and privacy, through an interdisciplinary lens that brings together scholars from governance, ethics, data science, urban planning, and engineering. Undoubtedly, these complex issues will need to be resolved to enable adoption of the UDI framework and/or other smart city analytical frameworks.

Conclusions

Data are increasingly becoming available on urban systems. Municipal officials, policy makers, and engineers are eager to adopt more data analytical approaches to uncover insights into sustainable urban planning and operations but face a major challenge in how to integrate spatially, temporally, and typologically heterogeneous urban data streams. The objective of this paper was to propose an urban data integration framework that is capable of integrating heterogeneous urban data in an extensible, scalable, and interpretable manner. The merits, applicability, and efficacy of the UDI framework was demonstrated on data from a midsize city in the United States (Palo Alto, California) and benchmarked against current practice (e.g., a relational database) for a typical urban analytics scenario. Results indicated that using the UDI framework enabled more-interpretable and computationally efficient data querying and exploration. In the end, by bridging the gap between currently disparate urban data streams, the UDI framework aims to provide a computational foundation in which smart city applications and data-driven approaches can be developed for cities. With the world rapidly urbanizing, new computational approaches to urban system design, management, and operations have the opportunity to make a tremendous impact on the lives of billions of people around the world.

Appendix I. Time Complexity of SQL Query

The approximated time complexity of executing the SQL based query for example in section "Data Querying and Retrieval" is calculated as follows:

Step 1. A loop is run on the buildings table. The complexity is $O(n_b)$:
- Step 1.1. For each building in buildings table, rows in the Building_Relationships table with object type = tree are found. The complexity is $O(n_{br})$:
  - Step 1.1.1. For each row selected in Step 1.1, the corresponding tree type from trees table is checked. The complexity is $O(n_t)$;
  - Step 1.1.2. If the tree type is oak, the corresponding row in the Building_Relationships table is returned;
  - Step 1.1.3. All the rows with tree type of oak are returned. If the number of rows returned is greater than four, the process is continued. Otherwise, the execution jumps to the next building;
  - Step 1.1.4. For the building selected in Step 1.1.2, all the rows in the building relations table with object type other than tree are selected;
- Step 1.2. The result of this part of the query is a table for buildings with at least four trees of type oak

Step 2. For the table generated in Step 1.2, the rows with proximate object type = traffic sensor are selected:
- Step 2.1. The corresponding sensor value in the Vehicle_Sensor Table is checked. The complexity is $O(n_c)$;
- Step 2.2. The time points for which the value is false are selected and returned;
- Step 2.3. The result of this part is a table with building name and time points which have to be checked;

Step 3. This step retrieves the energy consumption values and temperature values matching the table generated in Step 2.3:
- Step 3.1. The value of energy consumption is returned from the corresponding sensor type in the Smart_Meter table. The rows are only selected if the corresponding time points match the time points determined in Step 2.3. The time complexity is $O(n_e)$;
- Step 3.2. The value of temperature is returned from temperature sensor table when the time points match the up-to-the-minute time points determined in Step 2.3. The complexity is $O(n_{\text{temp}})$;

Step 1.1 and Step 1.1.1 involve a join between the Building_Relationships table and the Trees table. The time complexity of this join operation is given by $O(n_{br} \times n_t)$. This result is joined with the buildings table in Step 1, which gives the resultant complexity of $O(n_b \times n_{br} \times n_t)$. After Step 1 calculates the buildings which have at least five trees of type oak, this result is joined with the AOT_Temperature_Sensor table to find the time points for which there is no vehicle detected by the sensor. The resultant time complexity is given by $O(n_c \times n_b \times n_{br} \times n_t)$. Finally, this result is joined with the Smart_Meter to give the corresponding energy consumption values and then independently with the AOT_Temperature_Sensor table to extract the temperature values. The time complexity for these two joins is given as $O(n_c \times n_b \times n_{br} \times n_t \times n_e)$ and $O(n_c \times n_b \times n_{br} \times n_t \times n_{\text{temp}})$, respectively. Thus, the final complexity of executing SQL queries to retrieve data for the given urban analytics example is $O[n_c \times n_b \times n_{br} \times n_t \times (n_e + n_{\text{temp}})]$, which can be approximated as $O(n^5)$.

Appendix II. Time Complexity of Cypher Query

In the graph database, all the elements are linked by relationships and stored in memory. The approximated time complexity of executing the Cypher graph query for example in section "Data Querying and Retrieval" is calculated as follows:

Step 1. All buildings are traversed. The complexity is $O(1)$ for reaching each building;

Step 2. For each building, all proximate elements are checked to find all trees. The complexity is $O(n_t)$;

Step 3. All trees are traversed to find those with the type oak. The complexity is $O(1)$ for each tree;

Step 3.1. A count operation is performed to count the number of oak trees for each building and return the buildings if the number of oak trees is greater than 4;

Step 4. For the buildings returned in Step 3.1, values of the vehicle sensors are checked. If the value is false, the corresponding time points are returned. The complexity is $O(n_c)$:

Step 5. For time points returned in Step 4, the values of building energy consumption and temperature are also returned:

Step 5.1. For the energy consumption, the graph is traversed to reach the corresponding smart meter nodes and value relationships to obtain the values. The complexity is $O(n_e)$;

Step 5.2. For the temperature data, the graph is traversed to reach the corresponding ambient sensor nodes and value relationships to obtain the values. The complexity is $O(n_{\text{temp}})$;

The time complexity of executing Cypher queries to retrieve data from graph database for the given urban analytics example is: $O(n_t) + O(n_c) + O(n_e) + O(n_{\text{temp}})$, which can be approximated as $O(n)$.

Acknowledgments

The material presented is based in part upon work supported by the US National Science Foundation under Grant No. 1642315. Any questions, findings, and conclusions or recommendations expressed in the materials are those of the authors and do not necessarily reflect the views of the National Science Foundation. We also acknowledge the City of Palo Alto for their support and participation in this work. In particular, we thank Dr. Jonathan Reichental (CIO, City of Palo Alto) and Archana Gupta for their efforts on and support of this research and for providing feedback on the case study analysis.

References

Aguilera, U., D. López-de-Ipiña, and J. Pérez. 2016. "Collaboration-centred cities through urban apps based on open and user-generated data." *Sensors* 16 (7): 1022. https://doi.org/10.3390/s16071022.

Aljumaily, H., D. F. Laefer, and D. Cuadra. 2017. "Urban point cloud mining based on density clustering and MapReduce." *J. Comput. Civ. Eng.* 31 (5): 04017021. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000674.

AOT (Array of Things). 2018. "Array of Things, a networked urban sensor project that's changing our understanding of cities and urban life." Accessed March 15, 2018. https://arrayofthings.github.io/.

Balaji, B., A. Bhattacharya, G. Fierro, J. Gao, J. Gluck, D. Hong, A. Johansen, J. Koh, J. Ploennigs, and Y. Agarwal. 2016. "Brick: Towards a unified metadata schema for buildings." In *Proc., ACM Int. Conf. on Embedded Systems for Energy-Efficient Built Environments (BuildSys)*, 41–50. New York: Association for Computing Machinery.

Bocconi, S., A. Bozzon, A. Psyllidis, C. Titos Bolivar, and G. Houben. 2015. "Social glass: A platform for urban analytics and decision-making through heterogeneous social data." In *Proc., 24th Int. Conf. on World Wide Web*, 175–178. New York: Association for Computing Machinery.

Calabrese, F., L. Ferrari, and V. D. Blondel. 2015. "Urban sensing using mobile phone network data: A survey of research." *ACM Comput. Surveys* 47 (2): 25. https://doi.org/10.1145/2655691.

Castellani Ribeiro, D., H. T. Vo, J. Freire, and C. T. Silva. 2015. "An urban data profiler." In *Proc., 24th Int. Conf. on World Wide Web*, 1389–1394. New York: Association for Computing Machinery.

Chang, T. H., A. Y. Chen, C. W. Chang, and C. H. Chueh. 2014. "Traffic speed estimation through data fusion from heterogeneous sources for first response deployment." *J. Comput. Civ. Eng.* 28 (6): 04014018. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000379.

De Berg, M., M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. 2000. "Visibility graphs." In *Computational geometry*, 307–317. Berlin: Springer.

Doraiswamy, H., N. Ferreira, T. Damoulas, J. Freire, and C. T. Silva. 2014. "Using topological analysis to support event-guided exploration in urban data." *IEEE Trans. Visual. Comput. Graphics* 20 (12): 2634–2643. https://doi.org/10.1109/TVCG.2014.2346449.

Ferreira, N., J. Poco, H. T. Vo, J. Freire, and C. T. Silva. 2013. "Visual exploration of big spatio-temporal urban data: A study of New York City taxi trips." *IEEE Trans. Visual. Comput. Graphics* 19 (12): 2149–2158. https://doi.org/10.1109/TVCG.2013.226.

Goodchild, M. F., M. Yuan, and T. J. Cova. 2007. "Towards a general theory of geographic representation in GIS." *Int. J. Geograph. Inf. Sci* 21 (3): 239–260. https://doi.org/10.1080/13658810600965271.

Gu, Y., and C. Wang. 2013. "iTree: Exploring time-varying data using indexable tree." In *Proc., Visualization Symp. (PacificVis)*, 137–144. Piscataway, NJ: IEEE.

Han, L., T. Finin, C. Parr, J. Sachs, and A. Joshi. 2008. "RDF123: From spreadsheets to RDF." In *Proc., Int. Semantic Web Conf.*, 451–466. New York: Springer.

IBM (International Business Machines). 2012. *Smarter, more competitive cities: Forward-thinking cities are investing in insight today.* Somers, NY: IBM Smarter Cities.

Jain, R. K., J. M. Moura, and C. E. Kontokosta. 2014. "Big data + big cities: Graph signals of urban air pollution." *IEEE Signal Proc. Mag.* 31 (5): 130–136. https://doi.org/10.1109/MSP.2014.2330357.

Kang, T. W., and C. H. Hong. 2013. "The architecture development for the interoperability between BIM and GIS." In *Proc., 13th Int. Conf. on Construction Applications of Virtual Reality*, 30–31. Middlesbrough, UK: Teesside Univ.

Karan, E., A. Mohammadpour, and S. Asadi. 2016. "Integrating building and transportation energy use to design a comprehensive greenhouse gas mitigation strategy." *Appl. Energy* 165: 234–243. https://doi.org/10.1016/j.apenergy.2015.11.035.

Khan, Z., A. Anjum, and S. L. Kiani. 2013. "Cloud based big data analytics for smart future cities." In *Proc., 2013 IEEE/ACM 6th Int. Conf. on Utility and Cloud Computing*, 381–386. Piscataway, NJ: IEEE.

Kitchin, R. 2014. "The real-time city? Big data and smart urbanism." *GeoJournal* 79 (1): 1–14. https://doi.org/10.1007/s10708-013-9516-8.

Koperski, K., and J. Han. 1995. "Discovery of spatial association rules in geographic information databases." In *Proc., Int. Symp. on Spatial Databases*, 47–66. New York: Springer.

Langevin, J., J. Wen, and P. L. Gurian. 2015. "Simulating the human-building interaction: Development and validation of an agent-based model of office occupant behaviors." *Build. Environ.* 88: 27–45. https://doi.org/10.1016/j.buildenv.2014.11.037.

Lins, L., J. T. Klosowski, and C. Scheidegger. 2013. "Nanocubes for real-time exploration of spatiotemporal datasets." *IEEE Trans. Visual. Comput. Graphics* 19 (12): 2456–2465. https://doi.org/10.1109/TVCG.2013.179.

Lopez, V., S. Kotoulas, M. L. Sbodio, M. Stephenson, A. Gkoulalas-Divanis, and P. Mac Aonghusa. 2012. "Queriocity: A linked data platform for urban information management." In *Proc., Int. Semantic Web Conf.*, 148–163. New York: Springer.

Lund, P. D., J. Mikkola, and J. Ypya. 2015. "Smart energy system design for large clean power schemes in urban areas." *J. Cleaner Prod.* 103: 437–445. https://doi.org/10.1016/j.jclepro.2014.06.005.

Marique, A. F., S. Cuvellier, A. De Herde, and S. Reiter. 2017. "Assessing household energy uses: An online interactive tool dedicated to citizens and local stakeholders." *Energy Build.* 151: 418–428. https://doi.org/10.1016/j.enbuild.2017.06.075.

Miller, E. 1998. "An introduction to the resource description framework." *J. Lib. Admin.* 34 (3–4): 245–255.

Miller, J. J. 2013. "Graph database applications and concepts with Neo4j." In Vol. 2324 of *Proc., Southern Association for Information Systems Conf.*, 36. Atlanta: Southern Association for Information Systems.

Mitchell, W. J. 1990. *The logic of architecture: Design, computation, and cognition*. Cambridge, MA: MIT Press.

OGC (Open Geospatial Consortium). 2016. "CityGML." Accessed February 15, 2018. https://www.opengeospatial.org/standards/citygml.

O'Rourke, J. 1985. "Finding minimal enclosing boxes." *Int. J. Parallel Program.* 14 (3): 183–199. https://doi.org/10.1007/BF00991005.

Perini, K., and A. Magliocco. 2014. "Effects of vegetation, urban density, building height, and atmospheric conditions on local temperatures and thermal comfort." *Urban For. Urban Greening* 13 (3): 495–506. https://doi.org/10.1016/j.ufug.2014.03.003.

Raymond, C. M., G. G. Singh, K. Benessaia, J. R. Bernhardt, J. Levine, H. Nelson, N. J. Turner, B. Norton, J. Tam, and K. M. Chan. 2013. "Ecosystem service and beyond: Using multiple metaphors to understand human-environment relationships." *BioScience* 63 (7): 536–546. https://doi.org/10.1525/bio.2013.63.7.7.

Seedah, D. P., C. Choubassi, and F. Leite. 2015. "Ontology for querying heterogeneous data sources in freight transportation." *J. Comput. Civ. Eng.* 30 (4): 04015069. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000548.

Servigne, S., Y. Gripay, O. Pinarer, J. Samuel, A. Ozgovde, and J. Jay. 2016. "Heterogeneous sensor data exploration and sustainable declarative monitoring architecture: Application to smart building." In Vol. 4 of *Proc., ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, 99. Hannover, Germany: International Society for Photogrammetry and Remote Sensing.

Sheridan, J., and J. Tennison. 2010. "Linking UK government data." In *Proc., Linked Data on the Web Workshop*. Aachen, Germany: CEUR-WS.

Sinnott, R. O., et al. 2012. "A data-driven urban research environment for Australia." In *Proc., IEEE 8th Int. Conf. on E-Science (e-Science)*, 1–8. Piscataway, NJ: IEEE.

Snyder, J. P. 1987. *Map projections: A working manual*, Vol. 1395. Washington, DC: US Government Printing Office.

Sun, G., Y. Wu, R. Liang, and S. Liu. 2013. "A survey of visual analytics techniques and applications: State-of-the-art research and future challenges." *J. Comput. Sci. Technol.* 28 (5): 852–867. https://doi.org/10.1007/s11390-013-1383-8.

Sun, Y., and J. Han. 2012. "Mining heterogeneous information networks: Principles and methodologies." *Synth. Lect. Data Mining Knowl. Discovery* 3 (2): 1–159. https://doi.org/10.2200/S00433ED1V01Y201207DMK005.

Tollefsen, A. F., H. Strand, and H. Buhaug. 2012. "PRIO-GRID: A unified spatial data structure." *J. Peace Res.* 49 (2): 363–374. https://doi.org/10.1177/0022343311431287.

United Nations. 2014. "World's population increasingly urban with more than half living in urban areas." Accessed February 15, 2018. http://www.un.org/en/development/desa/news/population/world-urbanization-prospects-2014.html.

Van Hove, L. W., C. M. Jacobs, B. G. Heusinkveld, J. A. Elbers, B. L. Van Driel, and A. A. Holtslag. 2015. "Temporal and spatial variability of urban heat island and thermal comfort within the Rotterdam agglomeration." *Build. Environ.* 83: 91–103. https://doi.org/10.1016/j.buildenv.2014.08.029.

Venetis, P., A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu. 2011. "Recovering semantics of tables on the Web." *Proc. VLDB Endowment* 4 (9): 528–538. https://doi.org/10.14778/2002938.2002939.

Vukotic, A., N. Watt, T. Abedrabbo, D. Fox, and J. Partner. 2014. *Neo4j in Action*. Shelter Island, NY: Manning.

Wang, Q., and J. E. Taylor. 2015. "Process map for urban-human mobility and civil infrastructure data collection using geosocial networking platforms." *J. Comput. Civil Eng.* 30 (2): 04015004. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000469.

Wiemann, S., and L. Bernard. 2016. "Spatial data fusion in spatial data infrastructures using linked data." *Int. J. Geograph. Inf. Sci.* 30 (4): 613–636. https://doi.org/10.1080/13658816.2015.1084420.

Yang, Z., K. Gupta, and R. K. Jain. 2017. "A data integration framework for urban systems analysis based on geo-relationship learning." In *Proc., Int. Workshop on Computing in Civil Engineering*, 467–474. Reston, VA: ASCE.

Yuan, J., Y. Zheng, and X. Xie. 2012. "Discovering regions of different functions in a city using human mobility and POIs." In *Proc., 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 186–194. New York: Association for Computing Machinery.

Zheng, Y. 2015. "Methodologies for cross-domain data fusion: An overview." *IEEE Trans. Big Data* 1 (1): 16–34. https://doi.org/10.1109/TBDATA.2015.2465959.

Zheng, Y., H. Zhang, and Y. Yu. 2015. "Detecting collective anomalies from multiple spatial-temporal datasets across different domains." In Vol. 2 of *Proc., 23rd SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*. New York: Association for Computing Machinery.

Zhu, Y., and J. Ferreira Jr. 2015. "Data integration to create large-scale spatially detailed synthetic populations." In *Planning support systems and smart cities*, 121–141. New York: Springer.

Ziegler, P., and K. R. Dittrich. 2004. "Three decades of data intecration—All problems solved?" In *Building the information society*, 3–12. New York: Springer.

Zielstra, D., H. H. Hochmair, and P. Neis. 2013. "Assessing the effect of data imports on the completeness of OpenStreetMap—A United States case study." *Trans. GIS* 17 (3): 315–334. https://doi.org/10.1111/tgis.12037.