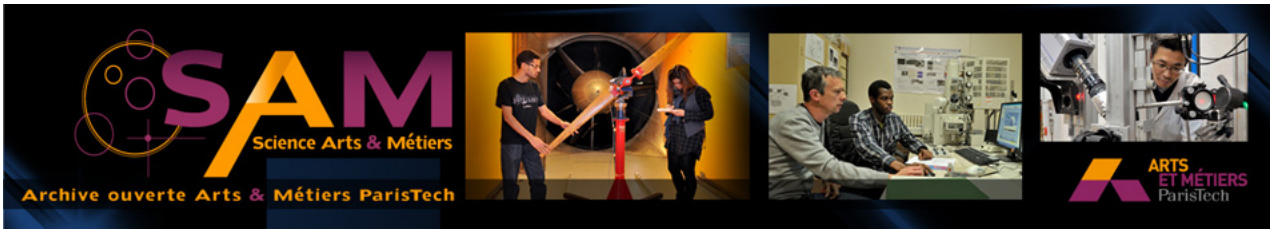
A PostgreSQL Extension for Continuous Path and Range Queries in Indoor Mobile Environments

Imad Afyouni, Cyril Ray, Sergio Ilarri, Christophe Claramunt



A PostgreSQL Extension for Continuous Path and Range Queries in Indoor Mobile Environments

Imad Afyouni[a,1,*], Cyril Ray[a,1], Sergio Ilarri[b,2], Christophe Claramunt[a,1]

[a] *Naval Academy Research Institute, 29240 Brest Cedex 9, France*
[b] *Department of Computer Science and Systems Engineering, University of Zaragoza, Maria de Luna 1, 50018 Zaragoza, Spain*

Abstract

Continuous location-dependent queries are key elements for the development of location-based and context-aware services. While most works on location-dependent query processing have been mainly oriented towards outdoor environments, this paper develops an approach for the continuous processing of location-dependent queries over indoor moving objects. A prototype for handling those queries has been developed as an extension for the open source DBMS PostgreSQL. Several algorithms for the continuous processing of path searches and range queries applied to both static and moving objects are performed on top of a hierarchical and context-dependent data model. Experimental results have been conducted to report our findings.

Keywords: Context-aware indoor navigation, location-dependent queries, hierarchical spatial data model, continuous processing, moving objects

1. Introduction

Mobile location-aware services have recently attracted extensive research attention as their development is expected to have significant impact for end users in both indoor and outdoor environments [Schiller and Voisard, 2004; Yu and Spaccapietra, 2010]. Location-aware services in indoor environments provide the user with the ability to interact with his/her physical surroundings in order to achieve some tasks. More generally, *context-aware systems* exploit contextual dimensions such as user-centred dimensions (e.g., user profile, user's physical/cognitive capabilities), the environmental context (e.g., location), the temporal context, and the context of execution (e.g., network connectivity, nearby resources). This allows to anticipate the user's needs and to customize his/her navigation experience [Baldauf et al., 2007].

Many indoor mobile applications need to incorporate a mechanism for the continuous processing of location-dependent queries over moving objects. Large indoor environments such as malls or commercial centres encompass a complex indoor infrastructure[3]. Examples of such services include continuous crowd monitoring within a given area, location-based alerts (e.g., continuously send E-coupons to all customers within 200 metres of a certain store), crowd notification in an emergency situation, and location-based friend finders (e.g., *"let me know if I am near a restaurant while any of my friends are there"*; and *"if I continue moving towards this direction, which will be my closest restaurants in the next 10 minutes?"*).

Location-aware, user-centred services can be distinguished according to two modes of data access: the *pull* mode depicts requests triggered by the user with the aim of pulling some location-dependent information from the service provider [Ilarri et al., 2010; Zhang et al., 2003], while the *push* mode represents services that are initiated by the service provider without having been requested by the user [Schiller and Voisard,

*Corresponding author
[1]E-mail: {imad.afyouni,cyril.ray,christophe.claramunt}@ecole-navale.fr
[2]E-mail: silarri@unizar.es
[3]For example, according to Shopinantes, the new Plaza Imperial Center in Zaragoza has an area of 159.000 square meters and 180 shops.

2004]. Location-dependent queries [Ilarri et al., 2010; Wang and Zimmermann, 2011], and more generally context-dependent queries, represent typical examples of pull-based services needed in such context-aware systems, as well as a key building block to detect situations of interest for push-based services. The context-dependent character of these queries means that any change in the context (e.g., changes in the locations and profiles of the objects that are involved in the query) may significantly affect the answer. For example, if a user wants to find out his/her friends within a range of 100 meters while navigating a shopping centre, the answer depends on both the user's current position and the location of the nearby friends. This type of query is particularly challenging because, in most cases, the user and the entities relevant for the query (e.g., the friends of the user) are moving.

An appropriate management of static and dynamic data is a key issue for processing location-dependent queries. In fact, the result of a query is only valid for a particular location of the query issuer and for certain locations of the objects of interest. As those queries are time-sensitive and location-dependent, they may be valid only for a given period of time (e.g., shops in a mall have certain opening hours and are not available outside that schedule), and within a given area. Therefore, those queries are expected to be processed as *continuous queries* [Terry et al., 1992], which means that the system should continually keep the answers up-to-date, until the query is explicitly cancelled by the user. Most work on location-dependent query processing has been developed for outdoor environments (cf., [Ilarri et al., 2010] for a recent survey). However, indoor environments have brought special features and constraints that should be considered during query processing. For instance, a hierarchical structure is typical in an indoor environment. Furthermore, small-scale indoor spaces encompass some specific properties, when compared with large-scale open spaces. In particular, interactions among humans and/or other objects in a small-scale space are much more frequent than in a large-scale one, which is usually beyond the range of humans' physical interactions. For indoor spaces, approaches for query processing based on the network distance are preferred and more realistic. However, existing approaches for network-based query processing usually assume an outdoor environment [Deng et al., 2009; Lee et al., 2005; Papadias et al., 2003], where for instance hierarchical networks do not appear.

The objective of the research presented in this paper is to process different kinds of location-dependent queries in a flexible manner, and to take into account additional context information, time-dependency, and the hierarchical layout of the indoor environment. In a previous work, a preliminary phase has been to develop an indoor data model that relies on a hierarchical and context-dependent design to allow a large spectrum of applications to be developed at different levels of abstraction, while alleviating performance and scalability issues in location-dependent query processing [Afyouni et al., 2013, 2010]. The present paper addresses the continuous processing behind other relevant location-dependent queries in indoor contexts. Particularly, continuous path searches and range queries represent the main focus of this paper. Those are briefly described as follows:

1. *Path queries* encompass all the queries that directly help the users to find and reach points of interest, by providing them with navigational information while optimizing some criteria such as the traversed distance or the travel time. Examples of such queries are: (i) discovering *optimal* paths to a nearest point of interest (e.g., landmark, place), and (ii) planning a path to a destination.

2. *Range queries* find and retrieve objects or places of interest within a user-specified range or area [Wu et al., 2006]. Those queries support navigation by continuously updating relevant details according to the users' movements. Ranges may be characterized by a circular or rectangular-shaped window in which objects of interest must be located. In addition, range queries may be static or dynamic according to whether or not the starting query point is in a static location. Similarly, a range query can be applied on static or dynamic data, depending on whether the objects that are the target of the query are moving or not.

An initial proposal presented in [Afyouni et al., 2012b] has discussed the principles of the continuous processing of path and range queries in indoor environments. This paper extends and improves this previous work by introducing a new prototype that has been fully implemented as a PostgreSQL extension for the continuous processing of location-dependent queries. Secondly, new algorithms, a series of optimizations, and more detailed explanations are provided. Thirdly, an experimental evaluation of the proposal has been

carried out, showing significant improvements thanks to the hierarchical and continuous query processing approach.

The remainder of this paper is organized as follows. Section 2 summarizes the hierarchical and context-dependent indoor data model, and emphasizes its role for location-dependent query processing. Section 3 introduces a hierarchical and incremental approach for the continuous processing of path queries over moving objects, while Section 4 presents an incremental approach for the continuous processing of range queries. Section 5 underlines the implementation experience by describing the PostgreSQL system architecture as well as several optimizations performed at the algorithmic level. Experimental results of the proposed solutions are provided in Section 6. Finally, Section 7 draws some conclusions and outlines future work.

2. Modelling Approach

The modelling approach summarized in this section has been introduced in a recent work [Afyouni et al., 2013], and represents a hierarchical graph representation of an indoor system that can be integrated into a context-aware system architecture. The preliminary requirements for the development of indoor spatial models have been surveyed in [Afyouni et al., 2012a] from a context-aware system perspective. An indoor data model should meet service-oriented (i.e., localisation, navigation, location-aware communication, activity-oriented interaction, and simulation and behavioural analyses) and efficiency-related (i.e., modelling effort, flexibility, performance, and scalability) requirements. This section first gives a brief summary of indoor space modelling approaches, and then presents a hierarchical and context-dependent indoor data model that is hierarchically organised and can be viewed as a tree structure in which location information is represented at different levels of abstraction. The proposed hierarchical data model constitutes the foundations on which query processing algorithms are performed.

2.1. Indoor Space Modelling Approaches

Modelling approaches are classified into two main classes: *symbolic* and *geometric* spatial models. Geometric-based approaches (otherwise referred to as metric or coordinate-based approaches) consider that locations are represented as points, lines, areas or volumes. In contrast, symbolic-based approaches provide qualitative human-readable descriptions about objects based on symbolic points of interest (e.g., room or floor identifier, building name, etc.). Symbolic-based approaches are often preferred, from an application perspective, over conventional geometric-based approaches, and have been used in many application scenarios [Becker and Durr, 2005], as they can capture the semantics of entities and places represented in an indoor space. For instance, deployment graphs have been proposed in Jensen et al. [2009] by using different types of positioning sensors in order to improve indoor tracking accuracy. Navigation between cells representing ranges of the deployed sensors is supported, which also allows for range-based analysis. However, the accuracy of location information in such techniques is relatively low since it depends on the sensor range. Furthermore, an optimised deployment strategy of the sensors is needed so that a more compact and more efficient graph can be created.

A generic data model for representing the complete movement of a moving object (i.e., indoor and outdoor movements), where the roads, streets and rooms are considered as constituting entities, has been also described in [Xu and Guting, 2011]. Specific data types have been provided for representing a room by a 2D area plus a value denoting the height above some ground level of the building. Besides, an indoor graph has been designed based on the *room* and *door* elements in order to support indoor trip planning at the room level. Doors in this graph represent nodes and edges correspond to rooms. Searches for optimal routes are made available through this model, based on a preprocessing step that computes paths between all pairs of doors. However, this model does not represent objects' movements at a fine-grained level. In addition, this approach does not deal with the continuous processing of current movements but rather with histories of movements.

Hybrid spatial models (i.e., with both geometric and symbolic representations) provide a good trade-off to efficiently integrate metric properties, while maintaining a more abstract view of space with easily-recognizable information about relationships between entities [Afyouni et al., 2012a]. A hierarchical data model embodies

knowledge of the environment at different levels of abstraction. A hierarchical design can support a large spectrum of applications, and offers a solution to alleviate performance and scalability issues in location-dependent query processing. Hierarchical models usually scale very well to large environments since queries such as path search can be performed hierarchically by switching from finer to coarser levels and vice versa. For more details about indoor space modelling approaches, please see [Afyouni et al., 2012a].

2.2. Preliminaries

This approach assumes that moving objects cooperate with a given system by providing up-to-date location data when needed. Thus, a minimum intervention of a user device is required for query processing by communicating the location of the user to the system according to a certain *location update policy* [Wolfson et al., 1999]. As soon as a location update is received from a moving object involved in a given query, the server starts the reevaluation process by considering the impact of such updates on the active queries. Accurate location data are assumed to be received in real-time from an indoor positioning system, based on recent technologies such as MEMS sensors, Wireless fingerprinting, and magnetic fields [Liu et al., 2007; Gu et al., 2009; Ray et al., 2010; IndoorAtlas, 2012].

For each location-dependent query, the following terms are used (as suggested in [Ilarri et al., 2006]). A *reference object* denotes an object that represents the reference for a given location-dependent query (e.g., for a range query, the object that indicates the centre of the range). A *target object* represents an object of interest to the location-dependent query, and which belongs to a specific *target class*. It is worth noting that no constraints are imposed on the movements and directions of the reference and target objects. Accordingly, a reference object is assumed to be either in a static location or moving freely in a spatial network with time-dependent edge weights. Similarly, a location-dependent query can request information about static or dynamic data, depending on whether the target objects are moving or not. For instance, a reference/target object could be a person or a point of interest -POI- (e.g., a room). Therefore, a unique combination of challenges arises, as the proposed architecture must be able to continuously process different kinds of location-dependent queries, and to take into account additional context information, such as the time-dependency and the user profiles (e.g., some areas may be restricted to special kinds of users, such as the security personnel), as well as the hierarchical layout of the indoor environment.

On the modelling side, the concept of *location granule*, first introduced in [Ilarri et al., 2011], represents a location at a given level of granularity (i.e., a node at the base level of the hierarchical data model, a room, a floor or a building). The idea is that it should be possible to express the queries and retrieve the results according to a given location granularity specified by the user. Location granules have an impact on: 1) the presentation of results; 2) the semantics of the queries; and 3) the performance of the query processing. A query language that takes advantage of these location granules to improve the expressiveness of location-dependent queries in indoor environments has been introduced in [Afyouni et al., 2013]. The query processing algorithms presented in Sections 3 and 4 constitute the main execution engine behind this query grammar, and allow for the continuous processing of those queries at different levels of granularity. Location granules allow to formulate queries with a location resolution which is appropriate for the intended application. With them, it is possible to formulate queries using the location terminology required by the user (e.g., vertices at the fine-grained level, rooms, floors, buildings, etc.). For example, a user may be interested in persons that are near the room where another (moving) object is currently located. In such a case, the location granularity is set to the room level.

2.3. Hierarchical Data Model

This section presents the hierarchical spatial component of the indoor data model. This component constitutes the foundations on which the algorithms for the continuous processing of location-dependent queries are performed. A spatial component $\mathcal{S} = \bigcup_{i=1...|\mathcal{S}|} \mathcal{S}_i$ is made of a set of layers ($\mathcal{S}_i$) hierarchically organised and representing the indoor environment. The core layer ($\mathcal{S}_1$) is hereafter presented. Then, other coarser layers that are incorporated into the query processing are discussed.

2.3.1. Core Spatial Layer

The core layer $\mathcal{S}_1$ (referred to as $\mathcal{S}_{micro}$) of the indoor data model is made of a fine-grained graph G_{micro} = (V_{micro}, E_{micro}, W_{length},W_{time}) embedded within a spatial grid with a regular cell size, and which covers the indoor space (Figure 1). The extent and the level of granularity are two mandatory parameters that have to be determined *a priori* for the derivation of the grid. The accuracy of the resulting grid depends on the cell resolution. A fine-grained grid supports accurate location data, but could introduce heavy processing workloads [Afyouni et al., 2012a]. For example, the spatial resolution selected in the scenario illustrated in Figures 1 and 2 is 50 cm, which roughly corresponds to the human spatial extent [Raubal, 2001]; this fine spatial resolution assumes highly accurate location data, but other coarser resolutions can be used depending on the application constraints. A coarser resolution results in a less accurate representation of space as well as a distorted perception of the objects' movements. The resulting grid graph encompasses vertices (i.e., nodes) that represent cells within the grid, and connections between cells are explicitly materialized by edges.

Each node is located at the centre of a cell, and is connected to its eight neighbours (not only the four ones located in the boundary) with horizontal, vertical, and diagonal edges. The size of cells affects the computation time as well as the accuracy of paths. A high resolution allows an optimal and accurate path to be computed, but consumes a higher computation time. In contrast, a big cell size favours better performance when computing navigational paths, but it is possible for narrow pathways to be missed in the modelling process. The choice of the appropriate level of granularity regarding the cell size is made at the application level in order to perform queries at the fine or coarse levels of granularity depending on the query objectives. For a context-aware navigation system, a fine resolution has been chosen so that accurate and optimal paths can be computed, and performance results are shown in Section 6 based on this resolution. Clearly, better performance results can be achieved by adopting a coarser cell resolution.

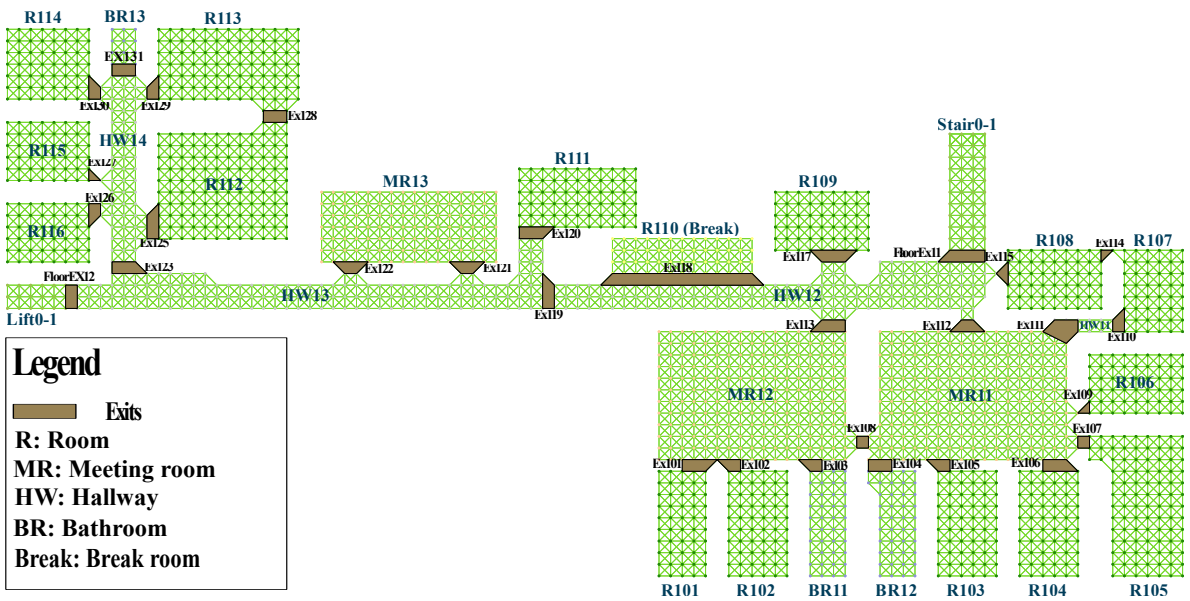


Figure 1: A fine-grained network of a two-storey building: first level of the hierarchical spatial data model

This modelling approach achieves a maximum coverage of the indoor space. An indoor environment is represented as a continuous space that supports continuous positioning techniques used in indoor navigation. Besides, the cell- and graph-based representation supports the modelling of structural properties (i.e., connections and relationships between nodes) at different levels of granularity, while keeping geometrical properties. Nodes of the grid graph are labelled according to their membership to a given spatial unit such as a room or a connecting space (i.e., a hallway). Therefore, each node has one and only one membership value since it belongs to one and only one spatial unit, whereas an edge might have multiple membership values

when it intersects several spatial units. Nodes and edges can be labelled with impedances defined at the application level (i.e., node's and edge's accessibilities). Figure 2 illustrates a closer view of the fine-grained network, and shows that a room at the abstract level may contain multiple nodes of the fine-grained graph (green points represent nodes, and links between them depict edges). The set of exits (i.e., brown polygons) illustrated in Figures 1 and 2 are considered afterwards (Section 2.3.2) in order to constitute an abstract layer as a part of the hierarchical data model.

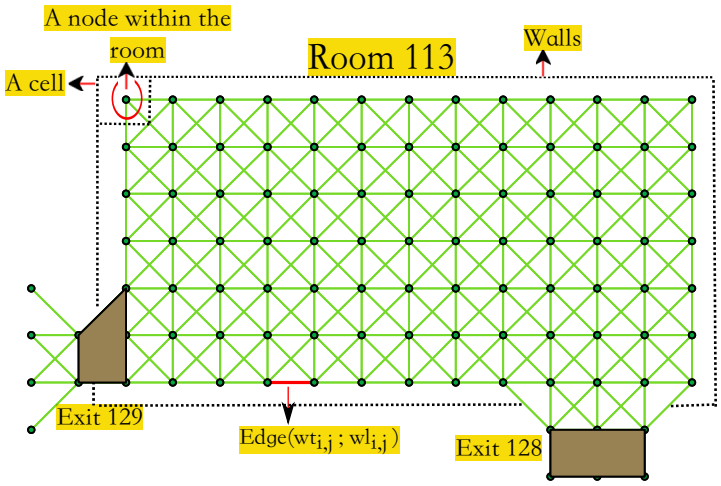


Figure 2: A closer view of the fine-grained graph (V_{micro}, E_{micro}) at Room 113 (R113) of the first floor. Each point represents a node located at the centre of a cell

$\mathcal{S}_{micro}$ is characterized by $V_{micro} = \{v_i\}$ as the set of vertices and $E_{micro} \subseteq V_{micro} \times V_{micro}$ as the set of edges. For each edge $e = (v_i, v_j) \in E_{micro}$, there exist two time-dependent cost functions $\omega l_{i,j}(t) \in W_{length}$ and $\omega t_{i,j}(t) \in W_{time}$ that compute the *length* and *travel-time* from v_i to v_j, respectively, if traversal is started at instant t. Besides time, this model also takes into account other contextual dimensions such as *user profiles* and *real-time events*, to further associate impedances with edge weights (cf., [Afyouni et al., 2013]).

Each node (or vertex) $v \in V_{micro}$ has a set of attributes that describe its physical location or state (i.e., whether it is accessible or not). A node v is formally defined by the tuple $\langle v_{id}, \mathtt{x}_v, \mathtt{y}_v, \mathtt{s}_v, \mathtt{L}_v \rangle$, where v_{id} is the node identifier, $(\mathtt{x}_v, \mathtt{y}_v)$ denotes the geometric location of v according to a local reference system, and $\mathtt{s}_v \in \{free, occupied\}$ determines whether or not the node v is physically occupied by an object at that moment, and $\mathtt{L}_v$ is a set of labels. Nodes that are occupied by static objects (or affected by real-time events that indicate that the node is inaccessible) are assumed to be unusable for path planning[4].

Let $\Sigma_{label} = \{\Sigma_{fine-grained} \cup \Sigma_{room} \cup \Sigma_{floor} \cup \Sigma_{building}\}$ be a set of labels or symbolic values that consists of all the identifiers of the topological hierarchy (i.e., local identifiers of nodes at the fine-grained level, as well as room, floor, and building identifiers) for a given space. Hence, $\mathtt{L}_v \subset \Sigma_{label} = \{$*local-id*, *room-id*, *floor-id*, *building-id*$\}$ is a set of labels assigned to v, where *local-id* denotes its local identifier at the fine-grained level, and the others are associated according to their belonging to the topological hierarchy. We assume at this level that v belongs to one and only one room, and one building. In contrast, *floor-id* is a subset of the set of floor identifiers since, for instance, a node located on a staircase may belong to several floors (Figure 1).

An edge $e \in E_{micro}$ is defined by a tuple $\langle (v_i, v_j), \mathtt{L}_e, \omega l_{i,j}(t), \omega t_{i,j}(t) \rangle$, where v_i,$v_j \in V_{micro}$, $v_i \neq v_j$, and $\mathtt{L}_e \subset \Sigma_{label}$ is a subset of the set of labels (e might have multiple labels when it intersects several spatial units -e.g., rooms-). Besides, $\omega l_{i,j}(t)$ and $\omega t_{i,j}(t)$ are time-dependent functions associated with the traversal of e. The traversal of some edges may be constrained by a temporal interval defined at the application level,

[4]Moving objects are usually not considered as obstacles, and even if they are obstructing the path (e.g., a cleaning machine blocking a pathway) they are expected to move in a short time. Nevertheless, there is no problem to model this kind of situations in our modelling approach, as a closed pathway can be considered as a real-time event that temporarily prevents passing by.

and within which the traversal is possible; otherwise the corresponding edge cannot be traversed. These functions are defined as follows:

$$\omega l_{i,j}(t) = \begin{cases} Ed(v_i, v_j) & if\ \mathtt{t} \in [t_{start}, t_{end}] \\ \\ \infty & otherwise \end{cases}$$

where $Ed(v_i, v_j)$ is the Euclidean distance between v_i and v_j, and t_{start} and t_{end} are defined at the application level (for example, $[08:00,\ \ 17:00]$ could be specified for an office building).

$$\omega t_{i,j}(t) = \begin{cases} f(\omega l_{i,j}(t)) & if\ \mathtt{t} \in [t_{start}, t_{end}] \\ \\ \infty & otherwise \end{cases}$$

where $f(\omega l_{i,j}(t))$ is a length-dependent time function that further associates impedances to compute the travel time between v_i and v_j.

The network distance and the travel time of a navigational path from v_s to v_d are computed as indicated in Definitions 1 and 2, respectively. These functions take the Euclidean distance derived from the fine-grained network in order to compute the optimal navigational network-based path, depending on either the distance and/or time criteria, as well as other semantic constraints. Therefore, the fine-grained graph implicitly integrates the notion of minimal indoor walking distance (as introduced in [Yang et al., 2010]) to compute navigational paths. This graph only includes accessible links from a given node, and links that intersect physical obstacles are directly removed at the modelling phase. Therefore, the Euclidean distance is only used at the very fine level to compute distances between cells. Navigational paths are computed based on accessible links and the resulting path represents the minimal walking distance from a source to a destination.

Definition 1. ***Fine-grained and time-dependent network distance:*** *Let $p = \langle v_{start}{=}v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k{=}v_{goal}\rangle$ be a path that contains a sequence of nodes $v_i \in V_{micro}$, i=1, ..., k. The time-dependent network distance of p is given by $length_{start,goal}(t_{start}) = \sum_{i=1}^{k-1} \omega l_{i,i+1}(t_i)$, where $t_i = t_{i-1} + \omega t_{i-1,i}(t_{i-1})$ represents the estimated time instant at node v_i, $\forall$ i=2, ..., k, and $t_1 = t_{start}$.*

Definition 2. ***Fine-grained and time-dependent travel time:*** *Let $p = \langle v_{start}{=}v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k{=}v_{goal}\rangle$ be a path that contains a sequence of nodes $v_i \in V_{micro}$, i=1, ..., k. The time-dependent travel time of p is given by $time_{start,goal}(t_{start}) = \sum_{i=1}^{k-1} \omega t_{i,i+1}(t_i)$, where $t_i = t_{i-1} + \omega t_{i-1,i}(t_{i-1})$ represents the estimated time instant at node v_i, $\forall$ i=2, ..., k, and $t_1 = t_{start}$.*

2.3.2. Coarser Spatial Layers

A coarse-grained model can be derived from a finer-grained representation depending on the application and context-aware constraints and capabilities. Our approach assumes that a given user acting in an indoor space can be continuously located in real-time, using for instance a MEMS sensor, thus providing a fine-grained representation for that user. In contrast, when WLAN or RFID positioning systems are deployed in the environment, a *coarser level* of granularity might be provided to locate users in the environment. Similarly, the spatial representation to consider in order to relate mobile users to the environment is chosen appropriately by taking into account some application constraints and properties. For instance, if one asks how many users are located in a given room, it may be inappropriate to display the precise locations of those users and their trajectories. In such a case, a representation at the *room level* is likely to be sufficient. Alternatively, a finer level of granularity might be appropriate when one is interested in the relative location of some users in a given room, and also with respect to the location of some sensors in that room. The relevant layers considered in the data model are summarized hereafter.

Exit hierarchy. Exits represent connections between rooms at the abstract level. An exit may also contain multiple nodes and edges at the fine-grained level (see Figure 1 and 2). Based on these exits, a coarser network (at a higher level of abstraction) can be designed, in which nodes depict those exits and links represents optimal navigational paths between directly reachable exits. An exit is an important element of the data model used for query processing. Through them, a user can leave or enter a place (e.g., doorways or staircases), and therefore the terms *exit* and *entrance* can be used interchangeably to emphasize the real

direction of movement. An exit is represented as an abstract node that belongs to two different spatial units (i.e., brown polygons in Figure 1). By means of these exits, optimal network distances and travel times between directly reachable pairs of exits are pre-processed and cached in order to reduce the cost of on-the-fly computation of hierarchical path searches. An exit ex' is directly reachable from exit ex if and only if there is an accessible passageway for pedestrians from ex to ex' which does not involve any other exit. Based on this concept, a network of exits/entrances, referred to as *exit hierarchy*, is constructed at a higher level of abstraction as illustrated in Figure 3. This layer is built on top of the fine-grained network represented in Figure 1.

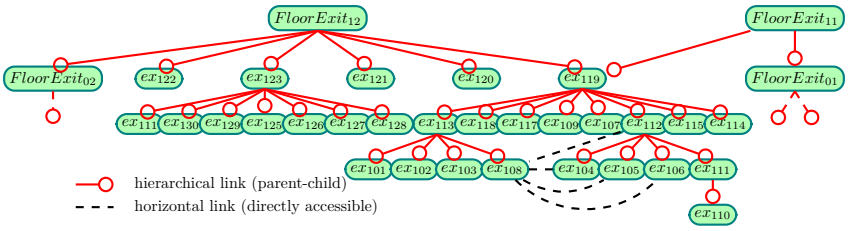


Figure 3: Part of the exit hierarchy derived from the fine-grained graph (first floor)

A link between two directly reachable exits is represented by a path (i.e., a sequence of nodes and edges in (V_{micro}, E_{micro})) at the fine-grained layer. More formally, let r, $r' \in \Sigma_{label}$ be the labels of two connected rooms, the exit representing the doorway between r and r' is given by: $ex_{r,r'} = \{v_i, v_j \in V_{micro} \mid \exists ex \in E_{micro}, ex = (v_i, v_j) \wedge r \in ex.\mathtt{L}_{ex} \wedge r' \in ex.\mathtt{L}_{ex}\}$. An exit is also characterised by: $\mathtt{L}_{ex_{r,r'}} = \{$*local-id*, $\{r, r'\}$, *floor-id*, *building-id*$\}$. $FloorExit_{11}$ is an example of an exit depicted in Figure 3, which belongs to two structural units: $Stair_{01}$ and HW_{12}. Therefore, $\mathtt{L}_{FloorExit_{11}} = \{$*FloorExit-11*, $\{Stair_{01}, HW_{12}\}$, *Floor-1*, *Building-1*$\}$. An abstract edge ($ex_{r,r'}$, $ex_{r',r''}$) in the exit hierarchy is a path made of a sequence of nodes and edges of the fine-grained level that compose the optimal network distance from a node $v_{start} \in e_{r,r'}$ to a node $v_{goal} \in ex_{r',r''}$. An edge of the exit hierarchy is referred to as *exit-path* and is denoted by $\langle source\_exit\_id, target\_exit\_id, length, time\rangle$. A generalisation of this hierarchy that covers a multi-storey building is used for path planning. Consequently, an exit of a ground floor has a *building exit* as a parent node, and a *first-floor exit* as a child node since both are parts of a staircase.

Location hierarchy. Incorporating information about exits into the topological hierarchy enables the modelling of optimal paths at an abstract layer. Those are used to facilitate hierarchical path searches and to alleviate performance issues raised while traversing the fine-grained graph. Although connectivity relationships between those elementary structural units can be computed from the exit hierarchy, an adjacency relationship needs to be associated to each unit in a separate abstraction layer. A room consists of a set of nodes at the fine-grained layer as illustrated in Figure 2. An abstract view of an indoor space considers rooms as abstract nodes and connections between rooms as links. Such topological properties are not explicitly materialised in the exit hierarchy, even though information representing their belonging to the topological hierarchy has been incorporated. Consequently, a location hierarchy that is based on a connectivity graph, which represents rooms as nodes and doorways as edges, can be automatically derived from the fine-grained graph as an additional layer in order to preserve topological relationships (Figure 4).
A room in the location hierarchy is characterized by $\langle room\_id, room\_type, Adj\_room\_list, \mathtt{L}_r\rangle$, where *room_type* describes whether this unit is a room, a meeting room, a hallway, etc., *Adj_room_list* denotes the list of identifiers of the adjacent units, and $\mathtt{L}_r$ is introduced in a similar way as in the fine-grained level. Such a location hierarchy can be directly derived from the fine-grained layer. A staircase that connects a given floor to another is represented as a room that belongs to the two corresponding floors, and which is bounded by two floor exits. On the other hand, an elevator is represented in a similar way to stairs. A multi-floor elevator consists of several stages that correspond to the number of floors of the building. Each stage of the elevator is modelled as a room that belongs to the two corresponding floors and bounded by exits/entrances to/from the corresponding floors.

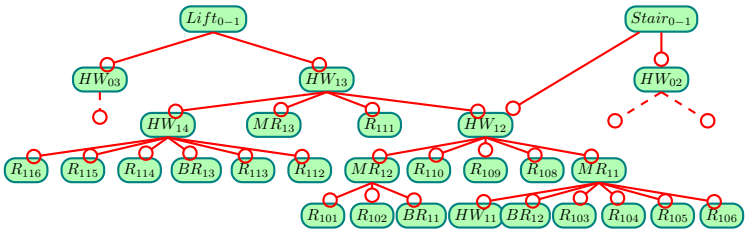


Figure 4: Part of the location hierarchy derived from the fine-grained graph (first floor); "HW" stands for Hallway, "MR" for Meeting Room, "R" for Room, and "BR" for Bathroom

From the fine-grained graph, a typical clustering process results in an abstract layer as illustrated in Figure 4. Graph partitioning is thus carried out based on the set of room labels associated to the nodes of the base graph. Consequently, this process consists of: (1) extracting and aggregating nodes whose room labels are identical to form the new abstract nodes of the location hierarchy; and (2) creating abstract edges between connected structural units, thus favouring topology-based queries. These steps are as follows:

- ***Step 1.*** Based on the set of room labels, the fine-grained graph is partitioned into subgraphs. Let $\varphi = \bigcup_{i=1\ldots|\Sigma_{room}|} \varphi_{\ell_i}$ be the set of subgraphs of S_{micro} such that $\ell_i \in \Sigma_{room}$, and where $\forall i \in \{1, \ldots, |\Sigma_{room}|\}$, $\varphi_{\ell_i} = (\mathtt{V}_{\ell_i}, \mathtt{E}_{\ell_i}) \subset S_{micro}$ is a subgraph extracted from the fine-grained graph according to node and edge labels, and where $\bigcap_{\ell_i \in \Sigma_{room}} \mathtt{V}_{\ell_i} = \emptyset$. An abstract node that represents each subgraph is then created, having ℓ_i as its *local-id*.
- ***Step 2.*** The set of outgoing edges between connected subgraphs is defined by: $\mathtt{E}_{\ell_i,\ell_j} = (\varphi_{\ell_i}, \varphi_{\ell_j})$ $\forall i, j \in \{1, \ldots, |\Sigma_{room}|\}$, $i \neq j$. It should be noted that, for geometric-based queries (e.g., path, range, and nearest neighbour queries), the exit hierarchy is more likely considered, as it lends itself to more accurate and more realistic pre-processing techniques. On the other hand, the location hierarchy is more suitable for topology-based queries (e.g., connectivity, adjacency, etc.) or when one looks for the optimal path that contains the smaller number of rooms. Therefore, there is no need to associate precomputed network distances to each edge in the location hierarchy.

Similarly, there exists a relationship between exit and location hierarchies since exits belong to structural units. For instance, by retrieving the list of room labels associated to all exits, one can derive connected rooms and rebuild the corresponding location hierarchy. Accordingly, switching between a location hierarchy and an exit hierarchy is always possible, thus covering a larger range of queries based on the different levels of granularity being managed.

3. Continuous Processing of Indoor Path Queries

In contrast to conventional location-dependent queries which consider static target objects, the algorithms presented in this paper seek maximum generality by assuming that the target objects can be either in a static location or moving freely in space. For simplicity, the query processing approach assumes that the reference object will follow its optimal path towards the target. Therefore, incremental search algorithms are required in order to efficiently execute continuous location-dependent queries, thus avoiding solving each search problem independently from scratch [Sun et al., 2009]. Incremental search implies reusing information from previous searches for each query to obtain the current result without having to recompute everything each time.

This section firstly discusses the most related work supporting path queries in indoor environments. Secondly, an overview on the approach for the continuous processing of path queries is given. Lastly, a detailed description along with pseudocodes of the corresponding algorithms are provided.

3.1. Background

Path queries implies finding an optimal route to a specified place or object of interest. One static/moving target object is considered is this kind of query. Few works have studied the problem of continuously processing path queries over moving objects located on a spatial network [Lee et al., 2007; Sun et al., 2010a,b, 2009]. The approach presented by [Lee et al., 2007] employs a mechanism to monitor the specified area for the continuous evaluation of fastest path queries. In addition, a grid-based index has been proposed to increase the efficiency of multiple query processing. Sun et al. have proposed a series of A*-based algorithms referred to as *Fringe-retrieving A** -FRA*- [Sun et al., 2009], *Generalized FRA** [Sun et al., 2010a], and *Moving target D* lite* [Sun et al., 2010b], that aim at providing an efficient incremental approach for moving target path search.

In particular, FRA* is an incremental version of A* that is applied on moving targets in grid maps, and aims at repeatedly finding the shortest path without having to process each search iteration independently from scratch. The algorithm regularly transforms the previous search tree to an updated tree based on the new locations of the reference and target objects. The current search tree is always rooted at the current location of the reference object. Each cell within the search tree maintains a pointer to its parent cell, so that a shortest path to the root can be directly obtained by traversing the tree in reverse following the ancestor nodes. Although this algorithm performs well on moving objects, its scalability to a large environment (i.e., a large campus with multi-storey buildings) is still an issue to address, since fine-grained grids have been adopted, which have not be proven to be scalable to large spaces [Afyouni et al., 2012a]. Moreover, FRA* does not take into account the hierarchical structure of an indoor environment, and thus cannot handle continuous path searches in multi-storey buildings.

On the other side, a hierarchical but static version of A*, referred to as *Hierarchical Path-finding A** (HPA*), has been proposed in [Botea et al., 2004], which decomposes a grid map into linked clusters and pre-computes optimal distances for crossing linked clusters at an abstract level. A bottom-up approach applied on a two-level hierarchy has been proposed. The technique is efficient, but has been used for path planning computations applied only on static data, and not to moving objects.

Those last two approaches have inspired our work on the continuous processing of general hierarchical path searches on moving objects described in Sections 3.2 and 3.3. Our approach transforms an initial search tree to an updated tree depending on the movements of the objects and the changes in the environment.

3.2. Algorithm Principles

Let us introduce an approach for the continuous processing of path queries that relies on a bottom-up technique, and which uses two levels of abstraction, that is, a fine-grained layer at the first level and the exit hierarchy at the second level of abstraction[5]. The search starts from a user-defined level of granularity (depending on the location granule specified in the request and which contains the initial query point) to the highest level of abstraction to find the optimal route at an abstract level. Refinement processes are executed, when needed, to find the exact location of the target object. The main steps of the process can be summarized as follows:

1. Find the optimal path within the initial granule until reaching the optimal exit.

2. Search at the abstract level (exit hierarchy) for the optimal path from the exit of the initial granule to the granule containing the target object.

3. Find the optimal path within the last granule to the target object, starting from the corresponding entrance of the granule.

4. Start a continuous path search by taking into account updated locations of both the reference and the target objects (considering the general case of moving targets). This implies transforming an initial search

[5]The location hierarchy presented in Section 2.3.2 could also be used if exact positions of objects of interest and accurate distances are not critically important for the user.

tree rooted at the previous location of the reference object to an updated tree rooted at its current location. The process continues by either expanding new subtrees from the leaves towards the target and/or by removing subtrees that are no longer needed.

Steps 1 to 3 represent the first iteration, which performs the hierarchical path search algorithm, presented in detail in Algorithm 1. Step 4 addresses the continuous processing of the query, which is presented in detail in Algorithm 2. Those are generic steps that may or may not be completely executed depending on whether the reference and the target objects are moving or not, and on the location granule specified in the query. Below is a typical example of a continuous path query that shows the applicability of the described approach:

```
SELECT gr('room-level', RO)
FROM Person AS P1, Person AS P2,
     All-routes(gr('micro-level', P1.id), gr('micro-level', P2.id)) AS RO
WHERE P1.id = 'userID1' AND P2.id = 'userID2'
MINIMIZE length(RO)
```

This query finds the shortest route from person 'userID1' to person 'userID2', showing the results at the *room* level. Notice that a location granule can be seen here as a cluster of nodes specified by the user and representing the current place (e.g., a room) containing an object. The *gr* operator expressed in this query can be referenced in the SELECT clause, the FROM clause and/or the WHERE clause of a query, depending on whether the granules are used for the visualisation of the results and/or for the processing of constraints or routes. For instance, SELECT gr('room-level', RO) is used to illustrate the sequence of rooms of the route obtained. On the other hand, *gr('micro-level', 'P1.id')* represents the fine-grained granule corresponding to the current fine-grained location of the object identified by 'P1.id' (cf., [Afyouni et al., 2013]). Two algorithms are described in the following section, which represent the implementation behind the *All-routes* operator used in this query.

3.3. Hierarchical and Incremental Path Search Algorithm

Algorithms 1 and 2 introduced in this section perform continuous path queries in two phases by taking advantage of the previously mentioned techniques. This approach is considered as a hierarchical and incremental version of A* applied to indoor moving objects. It is based on the hierarchical data model previously described in Section 2.3. Without loss of generality, a complex path query that requires performing all the steps described at the beginning of Section 3.2 is considered, which in the example given implies finding an optimal route from person p_1 to person p_2, assuming that p_2 is moving freely in space. Other cases, for instance where the target object is in a static location, can be easily tackled by performing a first iteration of the whole process and then skipping other unnecessary processing tasks (i.e., only Algorithm 1 would be executed). For the sake of clarity, handling granules is also not detailed in the pseudocodes. Scenarios where, for instance, granules at the room level are considered for the reference and/or the target objects, are easier to process and can be directly derived from this general scenario since no fine-grained network search is required.

As a variant of A*, our approach keeps two main data structures: 1) the CLOSED list contains exactly all the nodes that have been expanded (i.e., generated and added to the search tree); and 2) the OPEN list comprises all the nodes of the outer perimeter of the CLOSED list (i.e., outgoing neighbours of the leaves in the CLOSED list) that are not yet expanded. For each node v in the CLOSED list, the following properties are associated:

- The network distance from v to current location of the reference object v_{start} is computed, and referred to as the g-value $g(v)$; it holds that: $g(v) = g(parent(v)) + length_{parent(v),v}$, and $g(v_{start}) = 0$.
- An estimated heuristic value to the target node v_{goal} is applied, and referred to as the h-value, $h(v)$, which helps propagating a *"wavefront"* expansion towards the target node. $h(v)$ is computed as follows: $h(v) = Ed(v_{start}, v_{midway1}) + pathLength_{EX_s,Ex_g} + Ed(v_{midway2,v_{goal}})$, where $Ed(v, v')$ is the Euclidean distance between two nodes at the fine-grained level, $v_{midway1}$ and $v_{midway2}$ represent two midway nodes that belong to the reference and the target exits, respectively, and $pathLength_{EX_s,Ex_g}$

is the precomputed optimal network distance between the optimal exit at the start granule and the corresponding target exit. *pathLength*() is equal to zero if the reference room is equal to the target room or if the dimension of the *optimal exit path* is equal to one.

- In addition, each generated node is stored along with its path to the start node, a pointer to the predecessor node, the $f(v) = g(v) + h(v)$ value, and whether it has been expanded or not.

Notice that $g(v)$ and $h(v)$ are also time-dependent functions since they are computed by invoking the other time-dependent methods previously defined. The pointer to the parent node, $parent(v)$, is assigned in order to identify a reverse optimal path from the current node to the start node by following v's ancestors. The algorithm expands the node v with the smallest $f(v) = g(v) + h(v)$ from the OPEN list, and terminates when the OPEN list is empty or when the target node has been expanded.

Two main methods are frequently invoked during the execution of Algorithms 1 and 2. They are explained as follows:

- The *adaptedAstar*(*source*, *target*, *inPath*, *out outPath*, *out outLength*) method is used by the hierarchical path search for computing the fine-grained paths at the reference and the target granules. This method can also manage the different layers of the hierarchical data model. It can perform search either on the fine level separately or on the exit hierarchy to search for optimal exit paths. The main feature of this method is that it uses a *priority-queue-like* data structure which is indexed based on the value of $f(v)$ and represents the CLOSED list. A node with the minimum $f(v)$ is indexed on the top of the queue and thus retrieved first. Several information is dynamically inserted in the priority queue and used afterwards for the continuous processing. The *inPath* parameter of the *adaptedAstar*(...) method is used by the hierarchical path search to concatenate searches at different layers. Otherwise, *inPath* is considered to be NULL. An expansion procedure $expand(v)$ is performed throughout this method, and consists of checking for each neighbour v' of v whether it belongs or not to the OPEN list. If $v' \notin$ OPEN, the method generates v' by setting $g(v')$ to $g(v) + length_{v,v'}(t)$, setting its parent to v, and inserting it into the OPEN list. If v' is already in the OPEN list, then it checks whether $g(v') > g(v) + length_{v,v'}(t)$; if so, then the algorithm sets $g(v')$ to $g(v) + length_{v,v'}(t)$, and $parent(v')$ to v. The way the heuristic function is computed and other optimizations developed to improve the performance of the *adaptedAstar*(...) method are presented in Section 5.2.

- The $computeRefTarExits(vstart, vgoal)$ method is used for computing the *optimal exit path* that minimizes the path between the current locations of the reference and target objects. The result contains an optimal path at the exit hierarchy starting from the granule containing the reference object and ending with the corresponding entrance at the target granule. Notice that an optimal exit to a given object's location should not be necessarily the nearest one in term of distance, but rather the one that optimizes the whole network distance between the reference and the target objects. This function is invoked when applying the continuous processing of hierarchical path and range searches.

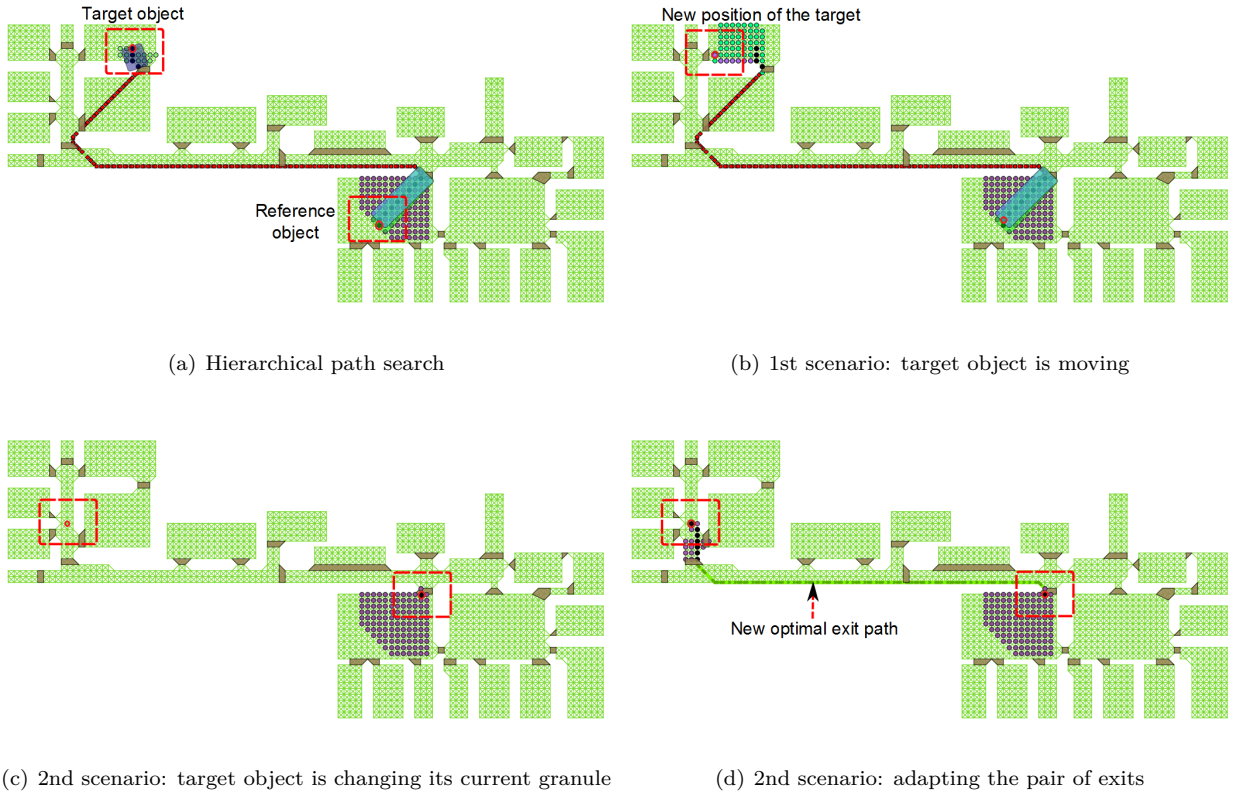
Hierarchical path search. The pseudocode of the hierarchical path search is illustrated in Algorithm 1. Figure 5(a) shows an example of a hierarchical path returned as a first path result between two moving objects. The main steps performed in this algorithm are explained as follows:

- The first part (*lines* $2 \rightarrow 4$) depicts the initialization of variables by involving the *queryLocation*() and *getNode*() methods, which return the current location of the object and the corresponding node in G_{micro}, respectively. $getDirectGranule(v)$ returns the identifier of the granule containing v. The *getNode* method computes the nearest node to the current location of the moving object.

- The *lines* $5 \rightarrow 8$ check whether the current granules of the reference and target objects match. In that case, no hierarchical search is needed, but instead an invocation the *adaptedAstar* method is performed, and a first result is returned. Otherwise, the $computeRefTarExits(vstart, vgoal)$ method is invoked to retrieve the best pair of exit/entrance that correspond to the source and target granules, along with the corresponding optimal exit path (*lines* $10 \rightarrow 11$).

Algorithm 1: *hierarchicalPathSearch(locRef, locTarg,* ***out*** *outPath,* ***out*** *outLength)*

Data: $\mathcal{S} : \bigcup_{i=1,2} \mathcal{S}_i : G_i = (V_i, E_i)$: hierarchical graph data; q: path query.
Result: A sequence of nodes of the optimal path $outPath = \{v_{start}, v_2, \ldots, v_{goal}\}$ to the target object where $v_i \in \mathcal{S}_1 \cup \mathcal{S}_2$, and the resulting network distance *outLength*
// $locRef/locTarg$: Reference/target object location; $g(v) = length_{v_{start},v}(t)$, $h(v)$, $parent(v)$: a predecessor is associated with each node and for each query; $CLOSED$: set of expanded nodes; $OPEN$: set of boundary nodes;
1 **begin**
2 $CLOSED \longleftarrow \emptyset$;
3 $v_{start} = getNode(locRef)$; $SGranuleId = getDirectGranule(v_{start})$; $OPEN = \{v_{start}\}$;
// A new position implies a new root of the tree;
4 $v_{goal} = getNode(LocTarg)$; $TGranuleId = getDirectGranule(v_{goal})$;
5 **if** $SGranuleId = TGranuleId$ **then**
6 $outRecord = adaptedAstar(v_{start}, v_{goal})$;
7 $outPath = outRecord.outPath$;
8 $outLength = outRecord.outLength$;
9 **else**
// Retrieve the best pair of exits of the source and target granules, and the corresponding optimal path;
10 $optimalExitPath = computeRefTarExits(vstart, vgoal)$;
11 $sourceExit = optimalExitPath[1]$;
// Step 1: Directed A* in G_{micro} from v_{start} to the source exit;
12 select a node $v_{midway1}$ such that $\{v_{midway1} \in sourceExit.nodeListIds \text{ and } v_{midway1} \in SGranuleId\}$;
13 $outPath = adaptedAstar(v_{start}, v_{midway1})$;
// $adaptedAstar$ removes v with $f(v) = g(v) + h(v) = \min_{v' \in neighbours(v)} f(v')$ from $OPEN$;
// And then inserts v into $CLOSED$;
// Step 2: Insert all exits of $optimalExitPath$ into $OPEN$;
14 $generate(sourceExit)$; $insert(sourceExit)$ into $OPEN$; $parent(sourceExit) = v_{midway1}$;
15 **foreach** *exit* $e \in optimalExitPath$ **do**
// All-pairs optimal network paths between exits are already precomputed;
16 $generate(e)$; $insert(e)$ into $OPEN$; $parent(e) = e'$; // e' is the predecessor of e;
17 **end**
// Step 3: Directed A* in G_{micro} until reaching v_{goal};
18 $targetExit = optimalExitPath[length(optimalExitPath)]$;
19 select a node where $\{v_{midway2} \in targetExit.nodeListIds \text{ And } v_{midway2} \in TGranuleId\}$;
20 $currentPath = append(outPath, optimalExitPath)$;
// The final $outPath$ is obtained by applying a $reversePath$ procedure from v_{goal} to v_{start} following v_{goal}'s ancestors;
21 $outRecord = adaptedAstar(v_{midway2}, v_{goal}, currentPath)$;
22 $outPath = reversePath(v_{goal}, v_{start})$;
23 $outLength = outRecord.outLength$;
24 **end**
25 **end**



(a) Hierarchical path search

(b) 1st scenario: target object is moving

(c) 2nd scenario: target object is changing its current granule

(d) 2nd scenario: adapting the pair of exits

Figure 5: Hierarchical and incremental path search algorithm

- **Step 1:** Once the optimal exit path is computed, the algorithm starts the hierarchical path search by firstly performing a fine-grained search at the reference granule until reaching a midway node that belongs to the source exit previously identified (*lines* 12 → 13).
- **Step 2:** Upon identifying and reaching the first exit of the computed optimal exit path, the algorithm moves up to the upper level at the exit hierarchy, and inserts all exits of the optimal exit path into the priority queue (i.e., CLOSED list), so that concatenated paths of two levels of granularity will be associated to each of those generated exits (*lines* 14 → 17). The $generate(e)$ methods sets $g(e)$ to $g(v) + length_{v,e}(t)$, and $parent(e)$ to v. This step continues until reaching the optimal entrance at the target granule.
- **Step 3:** The hierarchical path search algorithm ends by performing a fine-grained search starting from an identified midway node at the target granule towards the exact location of the target, and then by identifying a shortest path in reverse by following v_{goal}'s ancestors until reaching the source node (*lines* 18 → 23).

Continuous query processing. A continuous processing of path queries starts at this phase by taking into account the updated locations of the reference and the target objects. A pseudocode of this algorithm is illustrated in Algorithm 2. A description of this algorithm is given as follows:

- A fundamental step consists of invoking the hierarchical path search method previously described in order to build for the first time the search tree, which will be stored in the priority queue structure (*lines* 2 → 4). After this step, a first path result is returned to the user, and all generated nodes in the search tree rooted at v_{start} are associated with the above mentioned properties.

Algorithm 2: *continuousHPath(refObjId, tarObjId) returns SETOF [outPath, outLength]*

Data: $\mathcal{S} : \bigcup_{i=1,2} \mathcal{S}_i : G_i = (V_i, E_i)$: hierarchical graph data; q: path query; up-to-date location data of ref/target objects.

Result: A continuous set of optimal paths *outPath*, and the resulting network distance *outLength*

// $refObjId/tarObjId$: Reference/target object identifier; $CLOSED$: set of expanded nodes; $OPEN$: set of boundary nodes.

begin
  $locRef = q.queryLocation(refObjId)$; $v_{start} = getNode(locRef)$;
  $locTarg = q.queryLocation(tarObjId)$;
  $[outPath,\ outLength] = hierarchicalPathSearch(locRef, locTarg)$;
  // At this stage, a complete search tree has been built and stored;
  // Continuous path search (keeping the initial answer up-to-date);
  while $v_{start} \neq v_{goal}$ **do**
    previous-$v_{start} = v_{start}$;
    $v_{start} = getNode(q.queryLocation(refObjId))$;
    $v_{goal} = getNode(q.queryLocation(tarObjId))$;
    if *previous-*$v_{start} == v_{start}$ and $v_{goal} \in CLOSED$ **then**
      // The answer is returned without extra computation;
      $[outPath, outLength] = \{reversePath(v_{goal}, v_{start}),\ g(v_{goal})\}$;
    else
      if *previous-*$v_{start} \neq v_{start}$ **then**
        // An updated search tree is being created with a new root v_{start};
        $updateTreeRootedAt(v_{start})$; // *g-values* are not affected;
        $deleteUnnecessaryNodes()$; // Unnecessary nodes from $CLOSED$ are deleted;
        $completeOPEN()$; // Nodes of the outer perimeter are added;
      end
      if $v_{goal} \notin CLOSED$ **then**
        // Tracking of v_{goal}, check the new optimal pair <exit/entrance>;
        $newOptimalExitPath = computeRefTarExits(vstart, vgoal)$;
        if $(newOptimalExitPath == optimalExitPath)$ **then**
          continue A* in G_{micro} with the same $OPEN$ and $CLOSED$ lists until reaching v_{goal};
        else
          // v_{goal} is either nearer to another exit within the same granule or has left the last granule;
          delete subtree rooted at $LastExit$ from $CLOSED$;
          insert not generated exits from $newOptimalExitPath$ into $OPEN$;
          repeat Step 3 of Algorithm 1 starting from $parent(newOptimalExitPath[n])$ until reaching v_{goal};
        end
      end
      $[outPath, outLength] = \{reversePath(v_{goal}, v_{start}),\ g(v_{goal})\}$;
    end
    $sleepUntilNextPositionUpdate(minWaitingTime)$; // The thread remains asleep while no location update is performed or the minimum waiting time between iterations (if specified) has not been consumed
  end
end

- A continuous path search starts with the aim of keeping the initial tree up-to-date. At each iteration, the algorithm looks for up-to-date locations of the reference and target objects, and then matches those locations to nodes at the fine-grained network (*lines* $6 \rightarrow 8$). As long as the search tree is rooted at the same v_{start} (i.e., the reference object is not moving) and the target object is located on a node in the CLOSED list, a shortest path can be easily determined in reverse from v_{goal} towards v_{start} (*lines* $9 \rightarrow 10$).

- When the reference object moves (*lines* $12 \rightarrow 15$), additional steps to transform an initial search tree rooted at the previous v_{start} to an updated tree rooted at the current v_{start} are needed. Three main functions are invoked to perform this transformation: 1) The $updateTreeRootedAt$ method firstly updates pointers to parent nodes at the reference granule so that nodes of the reference granule are rooted at the new v_{start}; 2) secondly, the $deleteUnnecessaryNodes()$ method removes unnecessary nodes from the previous CLOSED list; and finally 3) $completeOPEN$ is called to add nodes of the outer perimeter of the new CLOSED list to the new OPEN list.

- In case v_{goal} is not located in the CLOSED list, a new invocation to the $computeRefTarExits()$ method is performed to determine the new optimal exit path towards the target (*lines* $17 \rightarrow 21$). If the new optimal exit path matches the previous one (i.e., this means that the same target exit is still the nearest one), the algorithm performs a directed search in G_{micro} with the same OPEN and CLOSED list until reaching v_{goal} (Figure 5(b)).

- Otherwise, the target is either nearer to another exit within the same granule or has left the last granule (*lines* $22 \rightarrow 25$). In that case, additional checks are performed to detect the *last common exit* between the new and previous exit paths. Once determined, the subtree rooted at that *Last exit* is no longer needed and will be removed from the CLOSED list, along with the nodes at the fine-grained level. Instead, a new subtree is created starting from the *Last exit* and by inserting exits of the new optimal exit paths, if any, until reaching the new optimal target exit. Finally, a similar search similar to the one performed in step 3 of Algorithm 1 is afterwards completed to reach the target (see Figures 5(c) and 5(d)).

- An optimal path is returned for each iteration from the current location of the reference object towards the current location of the target object. The $sleepUntilNextPositionUpdate()$ method is then invoked so that the thread remains asleep until the reference and/or target objects update their locations. Additionally, to keep the query processing overhead low in the presence of high location update rates, we may require a minimum time interval between iterations, by passing an optional argument $minWaitingTime$.

Notice that we are refreshing the answer periodically, as advocated in other works such as [Ilarri et al., 2006]. This is necessary because the answer will change all the time (even if slightly) due to the movements of the reference object and the target objects.

4. Continuous Processing of Indoor Range Queries

This section starts with a discussion of related work on continuous range queries in indoor environments. Next, we introduce our approach for the continuous processing of range queries which considers the mobility of both the reference and the target objects. This approach is based on a hierarchical range network expansion mechanism. The principle behind that approach is to continuously update the set of visited nodes that compose the range around the reference object. Furthermore, an indexed data structure referred to as *range queue* is built as a result of the hierarchical network expansion. Similarly to the priority queue structure described in Section 3.2, this structure maintains several properties associated to the generated nodes, such as the optimal path from the current node towards the source node. This particular property offers a significant advantage since it allows the system to provide not only information about whether an object is inside a specified range, but also to return a complete optimal path to that target object. Consequently, the result of

a continuous range query includes the set of qualifying object identifiers, their optimal path towards the reference object, and the corresponding network distance. A detailed description along with pseudocodes for the corresponding algorithms are hereafter provided.

4.1. Background

Range queries are used to retrieve information about objects or places within a specified range or area. Some range queries have a static reference object and others have a moving reference object. Similarly, the target objects of the queries can be static or moving. Recent works have studied location-dependent queries in indoor environments [Yang et al., 2009; Yuan and Schneider, 2010]. Specific graph data models that represent an indoor space have been designed in these approaches, thus allowing the processing of specific kinds of queries on top of the generated spatial network. These two related works are summarized as follows:

- In [Yuan and Schneider, 2010], the authors have introduced an approach to support range queries based on a virtual cell-based network generated for each query. Besides, an extension of this method has been proposed in the same paper to continuously process range queries whenever the query point (i.e., the reference object) moves. However, this approach is designed to address only range queries, and is only applied to static data (i.e., static points of interest). Moreover, for each query, a new virtual network that connects the query point to the predetermined points of interest is required, and additional computations are also needed to update the network each time the query point leaves its *safe area*. Furthermore, only information about whether static data are within the specified range is returned, without returning the optimal paths to those qualifying data.

- The method proposed in [Yang et al., 2009], is developed on top of a graph data model, and deploys a set of sensors to continuously monitor the users' movements, thus maintaining the query result up-to-date. The experimental results show that the provided data model is flexible, since it allows for different kinds of queries to be performed, and the solutions are efficient and scalable. However, the aim of this approach was to monitor indoor moving objects, so it only processes range queries over moving targets, but without taking into account a moving reference object as a starting point for the query. In addition, the model underneath relies on sensor-range-based positioning techniques, which is not perfectly suitable for navigation queries that may require fine-grained location information. Moreover, no information about the optimal path to the starting query point is obtained.

4.2. Hierarchical Range Network Expansion

A hierarchical network expansion is first computed starting from the location of the reference object in a similar way to the Dijkstra algorithm with multiple destinations and the Range Network Expansion algorithm [Papadias et al., 2003], see Figure 6(a). It consists of a "*wavefront*" expansion of the hierarchical network starting from the initial query point in all directions to find all nodes whose network distances to the source are less than the maximum specified threshold (i.e., the radius of the range query). The original idea here is that the valid routes are expanded hierarchically (cf. Figure 6(b)). The hierarchical network expansion mechanism is introduced in Algorithm 3 and takes into account the bottom-up approach explained in Section 3 to efficiently expand the valid routes within the specified radius. The main steps of the $hierarchicalNetworkExpansion(refObjId, radius, objectIds[])$ algorithm are described as follows:

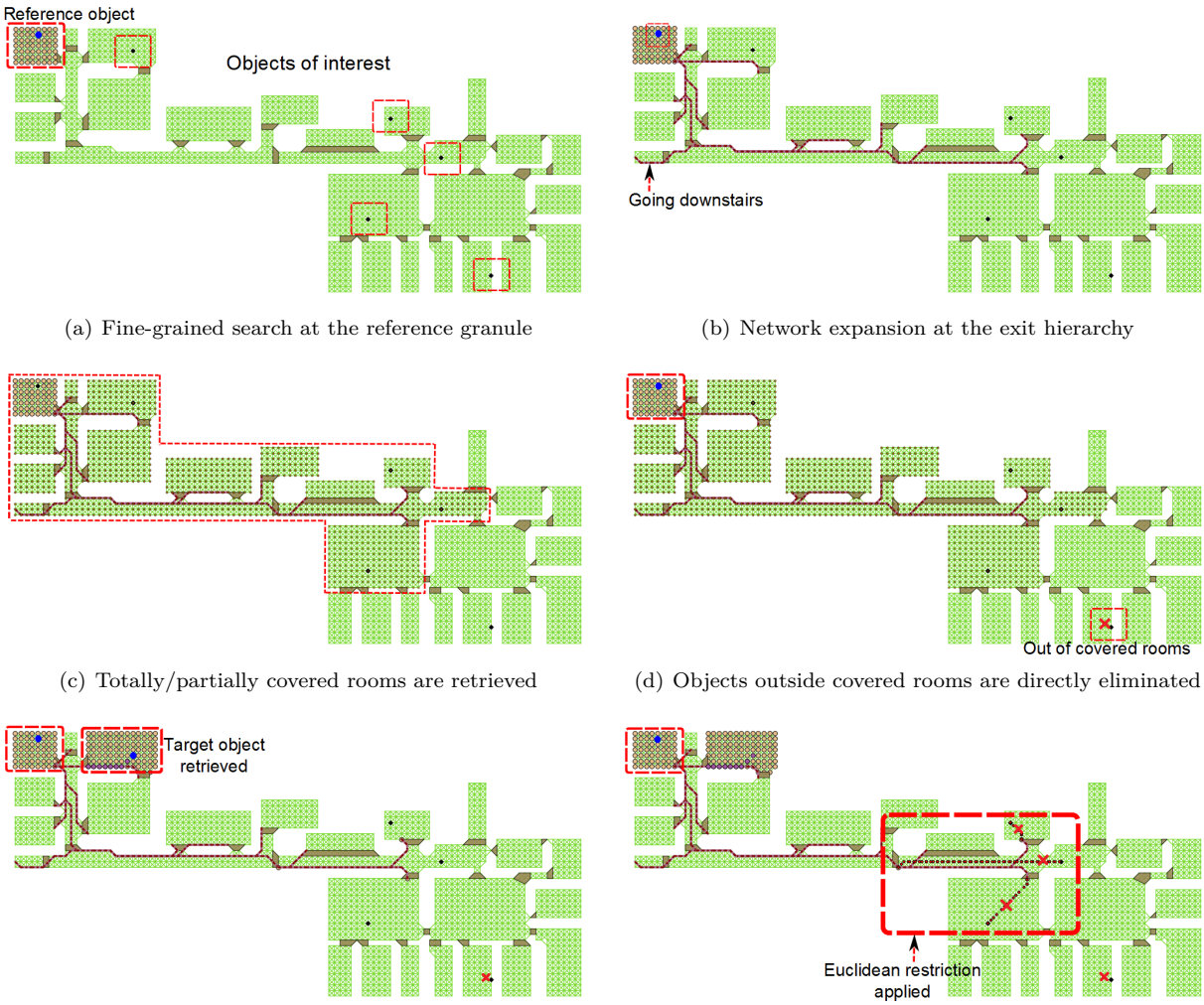
- *Step 1:* Gradually expand the valid routes in all directions within the initial granule of the reference object v_{start}, located at $LocRef$, while $length_{v,v'}(t_v) < radius$ ($lines\ 3 \rightarrow 4$). Nodes that are temporarily inaccessible or occupied by physical objects are automatically ignored. An internal invocation to the $networkExpansion(v_{start}, radius, inPath[])$ method is performed to execute this step. The range queue data structure is created for storing nodes accessible within the specified range. The $networkExpansion(...)$ method also handles expansion at different layers of granularity depending on the input values. When the network is expanded at the fine-grained level, only nodes that belong to the same granule as the reference object are expanded. Moreover, an $inPath$ parameter is also used to smoothly generate routes that are concatenated to previously expanded paths from earlier range

Algorithm 3: *hierarchicalNetworkExpansion(refObjId, radius, objectIds[])*

Data: $\mathcal{S} : \bigcup_{i=1,2} G_i = (V_i, E_i)$: hierarchical graph data; q: range query; up-to-date location data.
Result: *ResultSet*: Returns a SETOF [targObjID, outPath, outLength] for the qualifying target objects
// $C \subseteq objectIds[]$: `candidate set`; $LocRef$; $g(v)$; $parent(v)$; $RANGE$: `set of nodes around the reference object`; *coveredRooms*: `set of totally/partially covered rooms.`

```
1  begin
2      C ⟵ ∅; coveredRooms ⟵ ∅;
       // Step 1: At this stage, only nodes of the reference object's granule are expanded;
3      v_start = getNode(q.queryLocation(refObjId)); SGranuleId = getDirectGranule(v_start);
4      RANGE = networkExpansion(v_start, radius);// Network expansion only at the reference granule;

       // A new search tree RANGE is built after Step 1 and stored in the range queue;
       // Step 2: Network expansion at the Exit Hierarchy;
5      foreach accessible exit e ∈Exits of the reference granule do
6          select an expanded node v_midway1 in e;
7          RANGE = append(RANGE, networkExpansion(e, radius - g(v_midway1), v_midway1.path));
8      end
       // Step 3: Search for the qualifying objects;
9      foreach potentialQualifyingObject ∈ objectIds do
10         v_goal = getNode(q.queryLocation(objectIds[i])); TGranuleId = getDirectGranule(v_goal);
           // Totally/partially covered rooms have at least one accessible exit;
11         if objectIds[i] ∈ coveredRooms then
               // Apply Euclidean restriction at the target granule;
12             retrieve exit e ∈ TGranuleId that minimizes g(e) + Ed(e, v_goal);
13             if g(e) + Ed(e, v_goal) > radius − g(e) then
14                 display objectIds[i] is out of range;
15             else
                   // A fine-grained search at the target granule is required;
16                 select an expanded node v_midway2 in e;
17                 RANGE = append(RANGE, networkExpansion(v_midway2, radius − g(e), e.path));
18                 if v_goal is expanded then
19                     [targObjID, outPath, outLength] = {objectIds[i], reversePath(v_goal, v_start), g(v_goal)};
20                 else
21                     display objectIds[i] is finally out of range;
22                 end
23             end
24         else
25             display objectIds[i] is out of covered rooms;
26         end
27     end
28 end
```



(a) Fine-grained search at the reference granule

(b) Network expansion at the exit hierarchy

(c) Totally/partially covered rooms are retrieved

(d) Objects outside covered rooms are directly eliminated

(e) First object in range is returned

(f) Second object is eliminated by Euclidean restriction

Figure 6: Incremental algorithm for continuous range search: A range of 50 meters is applied in this example

searches (*inPath* takes NULL as a default value). In a similar way to Algorithm 1, an expanded node is stored along with its path to the reference object, a pointer to the predecessor node, and its g value.

- *Step 2:* On the exit hierarchy, start an expansion in all directions from the detected exits of the initial granule by taking into account the set of precomputed network distances (*exitPaths*) between directly reachable exits (*lines* 5 $\rightarrow$ 8). The expansion stops when no more exits can be added (i.e., when $g(e) \geq radius$). The resulting search tree includes all valid routes that consists of sets of vertices at two different levels of granularity. The rooms that are reachable from at least one entrance are considered as *covered rooms* (Figure 6(c)). Those covered rooms are determined to limit the search scope, so that target objects located outside this area are directly discarded.

- *Step3:* Search for the qualifying target objects by taking into account their up-to-date locations (*lines* 10 $\rightarrow$ 11). Different filtering processes are applied in order to avoid extra-computations resulting from searches at the fine-grained level (*lines* 11 $\rightarrow$ 15). The algorithm first discards an object if its current location is out the covered rooms (Figure 6(d)). For an object located within the covered rooms, it is checked whether the Euclidean distance between the optimal exit of the target granule

and its current location is greater than the radius. If the check is successful, the object will also be discarded for that iteration (Figure 6(f)). Otherwise, the algorithm proceeds by performing a network range expansion at the fine-grained level within the target granule until reaching all valid nodes that satisfy the specified threshold (*lines* 16 → 22). if v_{goal} has been discovered, a composite result that consists of a triple $\langle targObjID, outPath, outLength \rangle$ is returned (Figure 6(e)).

At the end of this process, all the generated nodes that constitute the valid routes within the radius are stored, along with their associated properties. The leaves, also referred to as *boundary nodes*, resulting from the range search are also returned. Such a hierarchical expansion provides a light way of exploring the network around the reference object and is performed just for the first time. It should be clarified that only exits and the paths between those exits are examined, but knowledge of nodes of the corresponding granules, which are not necessarily reachable with the same specified threshold, is not available. Therefore, all the corresponding granules of the valid exits are assumed to be accessible, but extra computations are required to determine, for each candidate object located at one of those nodes, whether that object really satisfies the distance constraint (i.e., to avoid false positives). This is done by computing the optimal path at the fine-grained level starting from the entrance of the target granule until reaching the target object.

4.3. Incremental Algorithm for Continuous Range Search

The algorithm introduced for the continuous processing of range queries in indoor environments applies an Euclidean restriction mechanism to retrieve candidate target objects that might be relevant to the final answer, as well as the hierarchical network expansion mechanism previously described. The continuous processing of range queries consists of:

1. Hierarchically expanding all the routes whose network distance from the source node is less than or equal to the specified radius. A hierarchical network expansion is performed once for the first iteration so that all the visited nodes within the range around the reference object are stored.

2. Continuously maintaining the set of parent nodes up-to-date when changing the root of the search tree (i.e., when the reference object moves). Boundary nodes are checked to decide, for each of them, whether to further expand that node or to perform a reverse search towards the source in order to remove nodes that are not relevant any more.

An example of such queries is: retrieve the identifiers of persons accessible at a network distance smaller than 100 meters from the room where object $o1$ is located:

```
SELECT Person.id
FROM Person
WHERE inside(100 meters, gr('room-level', 'o1'), Person)
```

Algorithm 4 illustrates the implementation of the *inside* constraint used in this kind of query. The algorithm is described as follows:

- Two functions are first invoked in *lines* 3 → 5. The first one applies the Euclidean restriction principle to retrieve candidate target objects, and the second one performs the hierarchical network expansion mechanism previously described in Algorithm 3.
- The first round of the algorithm returns a set of triples for the qualifying target objects. For the continuous processing, the main point is to update the set of parent nodes when changing the root of the search tree (i.e., when the reference object moves). There is no need to update all the distances to the new root. Instead, only distances and the parent pointers of nodes that belong to the granule of the reference object need to be rechecked, so that the tree rooted at the new position of the reference object is rebuilt (*lines* 10 → 12). This update allows to perform checks and to modify properties associated to the leaves as explained in the next step.

Algorithm 4: : *continuousRangeSearch(refObjId, radius, objectIds[])*

Data: $\mathcal{S} : \bigcup_{i=1,2} G_i = (V_i, E_i)$: hierarchical graph data; q: range query; r: network distance; up-to-date location data; $NetDistanceSet$.

Result: $ResultSet$: Returns a SETOF [targObjID, outPath, outLength] for the qualifying target objects

// $C \subseteq objectIds$: `candidate set;` $locRef$; $g(v)$; $parent(v)$; $RANGE$: `set of accessible nodes around the reference object;` $N \subseteq RANGE$: `set of boundary nodes,` $tempSet$: `temporary set of nodes.`

```
1  begin
2      C ⟵ ∅;
3      locRef = q.getRefObj.queryLocation(refObjId);
4      C = getObjectsInEuclideanRange(locRef, objectIds, radius);
5      RANGE = hierarchicalNetworkExpansion(refObjId, radius, C);
6      while NotCancel do
7          if locRef != q.getRefObj.queryLocation(refObjId) then
               // A new position implies a new root of the tree;
8              locRef = q.getRefObj.queryLocation(refObjId);
9              C = getObjectsInEuclideanRange(locRef, objectIds, radius);
10             foreach v ∈ RANGE and v ∈ getDirectGranule(getNode(locRef)) do
11                 UpdateParent(v);
                   // After this step, all the nodes in RANGE are rooted at the new locRef
12             end
13             foreach v ∈ N do
14                 length_{locRef,v}(t_v) = updateLength(v);        // reverse path search to locRef;
15                 if length_{locRef,v}(t_v) ≤ radius then
16                     tempSet = networkExpansion (v, radius − length_{v,locRef}(t_v), v.path);
17                     append(RANGE, tempSet);
18                 else
19                     v_current = parent(v);
20                     delete(RANGE, v);
21                     length_{LocRef,v_current}(t_{v_current}) = updateLength(v_current);
22                     while length_{LocRef,v_current}(t_{v_current}) > radius do
23                         v_current = parent(v_current);
24                         delete(RANGE, v_current);
25                     end
26                 end
27             end
28         end
29         foreach o ∈ C do
               // computePartOfPath will repeat steps similar to Step 3 in Algorithm 3;
30             v_goal = getNode(q.queryLocation(o));
31             if intersect(RANGE, getNode(o.LocTarg)) and
               computePartOfPath(e.path, getNode(o.LocTarg)) < radius then
32                 [targObjID, outPath, outLength] = {o, reversePath(v_goal, v_start), g(v_goal)}; // e is the
                   optimal target exit to o;
33             else
34                 display objectIds[i] is finally out of range;
35             end
36         end
37     end
38 end
```

- Only boundary nodes (i.e., leaves) of the RANGE list are checked to decide, for each of them, whether to further expand that node or to perform a reverse search towards the ancestors to remove nodes that are not relevant any more (i.e., the network distance to the new source is greater than the radius). For each boundary node, the algorithm first updates the network distance to the new source node, and checks whether that new distance is still less than the specified threshold (*lines* 13 $\rightarrow$ 17). If the check is successful, it completes the RANGE list by starting a new network expansion, and adds the valid nodes to the RANGE list. Otherwise (*lines* 18 $\rightarrow$ 25), it starts searches in the reverse direction from each boundary node, and removes nodes that are no longer needed from the RANGE list. This reverse search continues while the network distance from the current node towards the source node is exceeding the specified threshold. After this step, a new set of valid routes around the current position of the reference object is rebuilt.

- *Lines* 29 $\rightarrow$ 35 determine whether the target object is located on a node of the RANGE list. If so, the algorithm completes the partial path computation, as explained in Step 3 of Algorithm 3, starting from the optimal entrance of the target granule, and then checks whether that distance satisfies the specified threshold (i.e., $computePartOfPath(e.path, getNode(o.LocTarg)) < radius$). Target objects whose network distances to their current positions satisfy the maximum distance constraint are returned in the result.

5. Implementation

In this section, the PostgreSQL system architecture that has been developed is first presented, and then some optimizations implemented to improve the query processing algorithms are discussed.

5.1. PostgreSQL System Architecture

A database extension based on the open source DBMS PostgreSQL [Matthew and Stones, 2005] for handling continuous path searches and range queries has been implemented on top of a hierarchical network-based indoor data model. PostgreSQL is an object-relational DBMS (Database Management System) that allows implementing relational data models, new data types, functions, operators, triggers, etc. Several procedural languages are supported in PostgreSQL for developing functions and algorithms directly at the server side, so that the connection overhead and interprocess communication can be avoided. As a result, queries that are written as internal functions have the same access privileges and speed as native database functions and statements. The main parts of the prototype developed are:

- The hierarchical network-based data model of the indoor environments. Automatic methods to build the multi-storey fine-grained network and the time-dependent functions described in Section 2.3 have been developed. Methods to derive the exit and location hierarchies are also included.

- The operators and location-dependent constraints introduced in [Afyouni et al., 2013]. Those are implemented as PL/pgSQL functions applied on user-defined types.

- The algorithms to process continuous location-dependent queries over moving objects (cf. Sections 3 and 4).

The main advantage of implementing this prototype as a core database solution is that user-defined PL/pgSQL functions are used afterwards with native SQL statements to write location-dependent queries. However, the major problem encountered in implementing those functions was the lack of data structures supported. This required the use of temporary tables in the implementation of the priority and range queues. For testing, synthetic moving object datasets have been generated by using the Brinkhoff's network-based generator of moving objects [Brinkhoff, 2002], and then adapted to fit our needs.

5.2. Optimization

A series of optimization techniques have been employed to improve the efficiency of the proposed solutions. Those are explained with respect to their use in the path and range query processing as follows:

- A specific heuristic function has been developed and applied overall in the continuous path query processing, which tries to optimize network distance based on the hierarchical data model previously described. As mentioned earlier, a heuristic value is computed as follows: $h(v) = Ed(v_{start}, v_{midway1}) + pathLength_{EX_s,Ex_g}(t) + Ed(v_{midway2,v_{goal}})$. This heuristic function has been specifically designed to fit the hierarchical structure of the indoor environment. Consequently, a best estimation of the network distance towards the destination is taken into account during the expansion process, so that the node that minimizes the $gval + fval$ value is expanded first. The hierarchical-based heuristic function is used in the $adaptedAstar(...)$ method for directed path search, and in the $computeRefTarExits(...)$ method for computing the optimal exit path.

- An indexed *priority-queue-like* data structure for implementing the CLOSED list. A priority queue is characterized by a tuple $\langle vertexID, gval, fval, path, predecessor, expanded \rangle$, where $vertexID$ depicts the node identifier, $gval$ represents the network distance, $fval = gval + hval$ is an indexed parameter that is used as the priority measure to allow optimal network expansion, $predecessor$ contains the parent node, and the $expanded$ field depicts whether the node has been expanded or not. On the other hand, the range queue is indexed based on the $gval$, since no heuristic function is used in the network expansion mechanism. Those two data structures constitute the foundations on which the continuous processing for both algorithms is performed.

- Directional bounding boxes that help propagating a *"wavefront"* path search either towards the optimal exit or the target (cf. Figures 5(a) and 5(b)). Directional bounding boxes are considered as an important optimization of the $adaptedAstar(...)$ method. Those directional boxes limit the search for the neighbour nodes to those that are in the route direction towards the next goal. As a result, only five neighbours are generated and stored in the priority queue each time, instead of eight (the maximum number of neighbours). This reduces the execution time by 40%. A directional box is either oriented towards the next nearest exit or towards the target node if the reference and target objects are in the same room.

- The $computeRefTarExits(...)$ method computes, for path searches, the best pair of exit/entrance when the reference and/or the target objects move. For example, in Figure 1 let us consider $MR12$ and $HW14$ as a reference room (i.e., a room where the reference object is located) with five exits and a target room (i.e., a room where a target object is located) with seven exits, respectively. Next, a basic approach is to check all combinations of pairs of exits to determine the best pair. To optimize this process, an additional filtering process is developed in order to prune exits that do not have direct links to the target room and where no other open paths through them are available.

- For the continuous processing of range queries, two filtering techniques are employed, thus reducing the number of fine-grained network expansions at the target granules. First, totally/partially covered rooms are determined to limit the search scope, so that objects out of that search scope are directly discarded. Secondly, an Euclidean restriction at the target granule is applied to detect candidate objects that are far enough away from the reference object.

6. Experimental Evaluation

To the best of the authors' knowledge, no other work in the field of location-dependent query processing deals with hierarchical and continuous path searches and/or range queries on both moving reference and target objects in indoor environments. As mentioned in Sections 3.1 and 4.1, other approaches do not consider a multi-storey network, and in the case of range queries, either the reference or the target objects are assumed as static. Therefore, experimental results to evaluate the intrinsic properties of the proposed solutions are

presented in this section, since it is not possible to experimentally compare our approach with other proposals. In particular, a specific comparison between hierarchical and non-hierarchical processing approach is shown throughout these experiments. This criterion shows to what extent the continuous processing of a query is affected with respect to the mean execution time (in milliseconds), as well as the total number of expanded nodes in the search tree. The mean execution time shows the average CPU time of a continuous query answer for each location update. In contrast, the criterion about the total number of nodes shows the usefulness of the incremental processing approach by giving an indication of the global size of the search tree for a complete query evaluation.

To test the non-hierarchical configuration of both algorithms, two main methods have been developed. The $nonHierarchicalCPS(refObjId, tarObjId)$ method is an enhanced variant of the FRA* algorithm that integrates the multifloor settings as well as the time-dependent constraints for path computation. This method uses the *adaptedAstar* function to build a complete search tree at the fine-grained level, instead of making use of the hierarchical path search described in Section 3.3. This $nonHierarchicalCPS$ method applies techniques for the continuous path search similar to those applied in Algorithm 5, in order to update the search tree. Regarding continuous range queries, two main studies consider this problem in indoor environments [Yang et al., 2009; Yuan and Schneider, 2010]. The approach presented in [Yuan and Schneider, 2010] is only applied to static points of interest, and has not been experimentally evaluated. In contrast, the aim of the approach introduced in [Yang et al., 2009] was to monitor indoor moving objects, so it only processes range queries over moving targets, but without taking into account a moving reference object as a starting point for the query. The implementation of that work as a pure database solution is not straightforward, since the underlying graph data model depends on the deployment of sensors in the environment, and it is completely different from ours. In addition, not all the details of the technique are explained in the paper, so it is difficult to repeat their experiments as a database solution for comparison. Therefore, the $nonHierarchicalCRS$ method that uses a $nonHierarchicalNE$ method for the network expansion at the fine-grained level has been implemented. The performance of these methods is evaluated in the following sections with respect to the solutions proposed in Sections 3 and 4.

6.1. Experimental Settings

Two different system architectures can be applied for query processing. The former considers a server-based query processing architecture (either centralised or decentralised as discussed in [Afyouni et al., 2013]), where moving objects cooperate with the system by providing up-to-date location data (and possibly other information) when needed. Thus, a minimum intervention of a user device is required for query processing by communicating the location of the user to the system according to a certain location update policy [Wolfson et al., 1999; Ilarri et al., 2010]. The latter applies a client-based mobile architecture in which query processing is fully performed at the mobile device and locations of objects of interest are retrieved from the server. The first scenario implies more communication overhead, while the second scenario requires mobile devices with advanced processing capabilities. The first approach is adopted in our experimental settings, but nothing prevents testing those algorithms on a client-based mobile architecture. In the following section, scalability as well as performance testing are evaluated.

The experiments have been carried out on a MacBook Pro machine with a 2.3 GHz Intel Core i7 CPU and 4GB of RAM DDR3, and which runs Mac OS X 10.8.3. The PostgrSQL version used is 9.1.8 with the default settings. All tests were run 10 times in a completely independent way, and we verified that the individual results were consistent. The fine-grained two-storey network used for prototype evaluation consists of 4146 nodes and 13963 edges. The scenario considered for the performance evaluation retains a fine resolution of the fine-grained network by using a 50 cm distance between horizontal and vertical neighbour nodes. Other experiments showing the impact of different cell sizes are not illustrated due to space constraints. A coarser resolution provide better performance results, but with less accurate representation of space and objects' movements. Query performances increase dramatically with an increase of the cell size and particularly for queries performed at the room level (up to 60 times faster than the results shown with 50 cm cell size).

Due to the lack of real indoor moving object data, a synthetic dataset of around 1000 moving objects have been generated to evaluate how the prototype would behave in realistic scenarios. We use the Brinkhoff's network-based generator [Brinkhoff, 2002], which is suitable for all kinds of spatial networks. It is a generic

framework that can be adapted to specific scenarios. Indeed, the original purpose of this generator was to deal with moving objects on road networks. Hence, this generator does not directly deal with 3D network models, as the third dimension is not taken into account. Consequently, moving object data have been generated for each floor separately. The fine-grained graph of our model has then been integrated within the generator and some parameters have been tuned to generate indoor moving objects with realistic movements. Moreover, *a-posteriori* adaptations to the data set have been performed in order to take the multi-storey settings into account, and to transform the location data to the relative coordinate system considered in our scenario. In particular, we adapted some trajectories to simulate moving objects that move from one floor to another. Two additional methods have also been developed: i) the $inverseTransformation()$ function computes an inverse transformation to obtain coordinates in our referential system; and ii) the $computeNearestNode(xCoordinate, yCoordinate)$ determines the nearest node to a given moving object position.

The generator takes two input files (i.e., *.node* and *.edge* network data files) which correspond to the nodes and edges of the fine-grained network of the spatial data model. Different configurations have been adopted to define the mobility patterns of the generated moving objects. The duration of the evaluation period was set to 1000 timestamps. The waiting period between two successive timestamps was set to 1 second. The entire evaluation period is estimated to be around 15 minutes. Every moving object reports its motion parameters (i.e., location update, current speed) with a probability of 80% for each timestamp within the evaluation period. Objects move on the network at different speeds, with a maximum speed limit equals to 4 km/h. We choose some objects randomly and consider them as reference and target objects for both path and range queries.

6.2. Experimental Results

The following experiments first evaluate the continuous path search while varying the distance parameter between the reference and the target object. Secondly, a performance evaluation of the continuous range query processing is performed with respect to the number of objects and the radius parameters.

Continuous path searches. The first set of experiments shows how the continuous path search can be affected by applying a hierarchical or non-hierarchical-based query processing. The estimated distance between the reference and the target object is varied to demonstrate its impact with respect to both the average CPU time and the number of nodes expanded (see Figures 7(a) and 7(b), respectively).

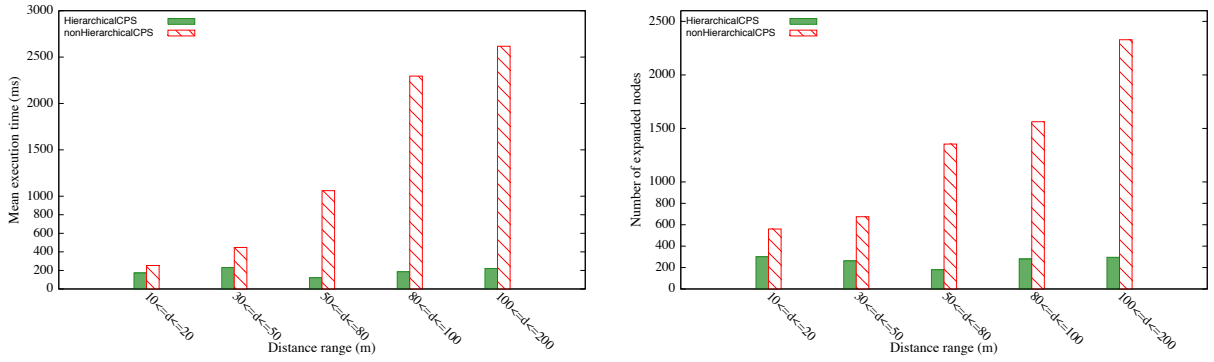
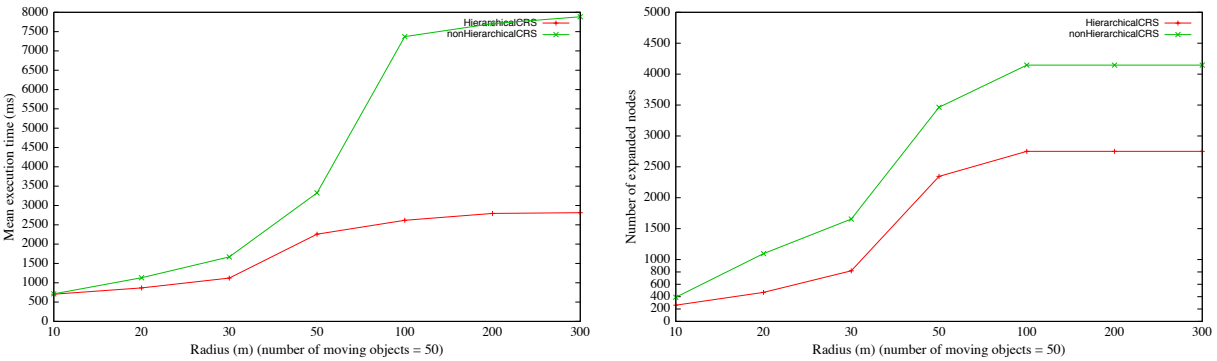


Figure 7: Varying the distance parameter: Hierarchical vs. non-Hierarchical Continuous Path Search

Figure 7(a) illustrates the mean execution time of a continuous path search with a distance range that varies from 10 to 200 meters. The same continuous processing techniques were applied for both hierarchical and non-hierarchical configurations. The results shows that the hierarchical approach keeps constant time responses when the distance between the reference and the target object increases. On the contrary, a

non-hierarchical configuration appears to grow with the distance between the reference object and the target objects. This is due to the fact that the hierarchical method processes fine-grained searches only at the reference and target granules, and thus whatever the distance between the two moving objects the time processing remains constant. We also performed tests with other distance values (not shown for the sake of clarity) and observed a similar behaviour. On the other hand, a relatively large distance between the two moving objects implies exploring a big part of the fine-grained network until reaching the target. This is clearly reported in Figure 7(b). As illustrated, a path search between two moving objects whose distance is between 100 and 200 meters, requires expanding around 200 nodes when applying the hierarchical approach, and around 2400 nodes with a non-hierarchical configuration. This demonstrates that the hierarchical and incremental path search algorithm is scalable to large indoor spaces (e.g., several multi-storey buildings in a campus) with constant time responses.

Continuous range queries. For the continuous range query algorithm, some parameters such as the range and the number of moving objects can be varied. This aims to show the scalability of the system and the algorithm behaviour over time. A performance comparison between two different scenarios based on either a hierarchical or a non-hierarchical network expansion mechanism is also considered. The next set of experiments studies the impact of varying the radius of a range query while setting the number of moving objects to 50 (see Figures 8(a) and 8(b)).



(a) (b)

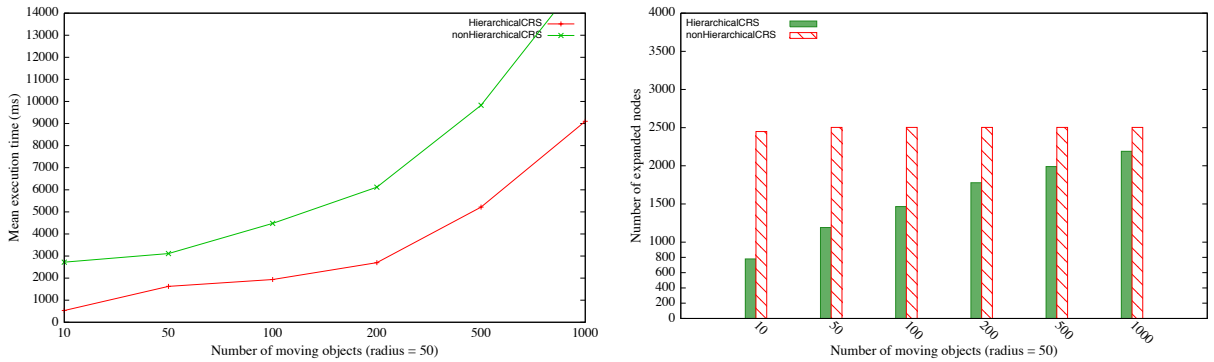
Figure 8: Varying the radius: Hierarchical vs. non-Hierarchical Continuous Range Search

The hierarchical network mechanism takes advantage of the exit hierarchy to explore the search scope without having to expand all the nodes in range at the fine-grained level. This means that this mechanism is dependent on the number and the locations of the target objects in the search scope. Wherever a candidate object requires a fine-grained search, the algorithm will explore the granule containing that object to decide whether it is is really within the range. Other granules in the search scope are not going to be expanded while no candidate objects enter those granules. On the other hand, a non-hierarchical network expansion is completely dependent on the search space. It expands all nodes within the range without taking into account the number and locations of the candidate objects. Notice that for the hierarchical approach a constant time is obtained after reaching a certain radius. This means that once the corresponding granules have been explored no extra-computation is required.

Moreover, testing for both configurations shows gains with respect to the number of nodes expanded during the whole process. Again, the total number of expanded nodes reaches a maximum for the hierarchical approach once the corresponding granules have been explored. On the contrary, the non-hierarchical approach reaches the maximum number of nodes of the network being considered in those experiments (i.e., it explores all the network).

The next set of experiments presents the performance evaluation and scalability of the two configurations with respect to the number of moving objects (see Figures 9(a) and 9(b)). In these experiments the radius

parameter is set to 50 meters, while the number of moving objects varies between 1 and 1000. The results shows significant improvement when applying the hierarchical processing, and an acceptable execution time even with 1000 moving objects. We should remind that this query returns, on each timestamp and for each moving object in range, the optimal path to the reference object. Consistently with the first result, Figure 9(b) shows that, whatever the number of moving objects specified by the user, the non-hierarchical configuration explores all the nodes within the specified radius. On the contrary, even with 1000 moving objects, the hierarchical approach is able to answer the query with a much smaller number of expanded nodes.



(a) (b)

Figure 9: Varying the number of moving objects: Hierarchical vs. non-Hierarchical Continuous Range Search

6.3. System Scalability

PostgreSQL supports full parallelism at the client-side, so that applications can open multiple database connections and manage them asynchronously, or via threads. Multi-thread Java programs with a connection pooling mechanism have been developed in order to simulate a multi-user environment, and to show the effect of concurrent continuous queries on the performance of the system (in this scenario, a single multi-core PostgreSQL server). We investigate the average response time of a continuous query per user. The response time considered in these tests is the average time interval between issuing a continuous query and getting the response from the system at a given timestamp when the search is successfully completed.

Figure 10 illustrates the average response time for a continuous path query at a given timestamp for a given user. The number of concurrent users querying the system in real-time varies from 1 to 200. Simulation results suggest that, with 30 to 50 concurrent accesses, the average response time varies between 1 and 1.5 seconds. Even with 100 to 200 concurrent path queries, the time for a query answer remains acceptable, and the number of concurrent queries has a linear impact on the performance (it should be noted that the X-axis in the figure is not linear).

The results of the experiments that consider concurrent range queries are illustrated in Figures 11(a) and 11(b). As may be expected, a range query results in heavy processing costs, which have been noted by previous tests (Figures 9(a) and 9(b)). Therefore, the simulation shows a reduced number of concurrent accesses that varies from 1 to 50.

Two types of experiments have been performed. On the one hand, Figure 11(a) illustrates the average response time while varying the number of concurrent users, and with two different thresholds: 30 and 50 meters. Simulation results show a good performance of the average query answer with up to 30 concurrent users, and an acceptable time response with up to 50 users. On the other hand, Figure 11(b) shows the performance evaluation of a concurrent range query with different numbers of moving objects: 20, 40 and 60. The system shows a good scalability with up to 50 concurrent users when the number of objects specified is low. With a bigger number of moving objects, the system can report good response times with up to 30 users.

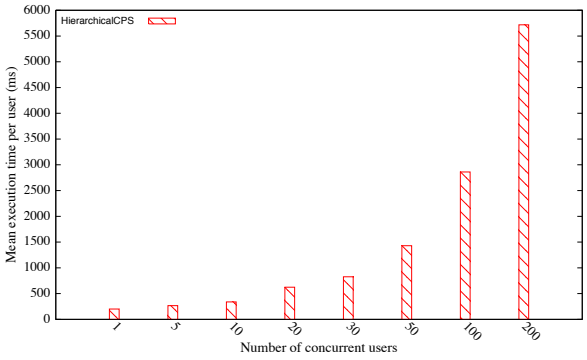
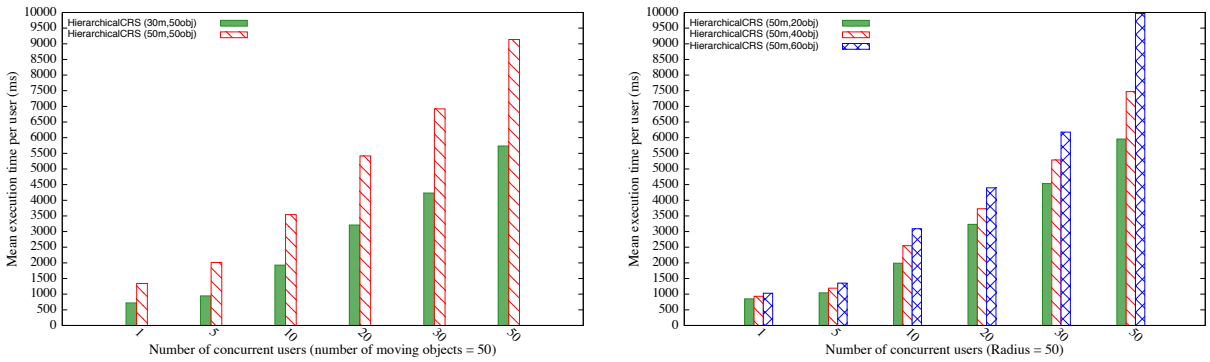


Figure 10: Varying the number of concurrent access: Hierarchical Continuous Path Search



(a) (b)

Figure 11: Varying the number of concurrent access: Hierarchical Continuous Range Search

6.4. Summary of the Experiments

With respect to the above experimental results, the execution of the algorithm developed for continuous path search appears as satisfactory regarding execution time and scales well with the number of expanded nodes. It has been shown to be scalable enough to large indoor spaces thanks to the hierarchical-based query processing. Moreover, the continuous range query processing approach provides satisfactory scalability with respect to the radius parameter, and acceptable performance in processing range queries when the number of moving objects increases. Regarding the experimental results for the continuous range search, all moving objects involved are assumed to be of interest to the corresponding query. Actually, only objects of a certain type (the ones involved in the query) have a direct impact on the performance of the query processing, so this generates a worst-case situation. A pre-filtering of objects based on static properties (e.g., people in my friend list) has a similar effect, as this reduces the number of objects to consider as a potential candidate (i.e., moving objects not in the friend list are immediately discarded). Consequently, and for example, the largest mean execution times shown in Figure 11(b), when applied to a friend-finder application, would imply a range of 50 meters and 60 persons in the list of friends that should be at the same time in the same indoor environment. Nevertheless, the total number of moving objects, independently of their type, has also a slight impact on the performance of the server due to the need to manage their location updates.

Furthermore, the whole system has been tested for scalability with the respect to the number of concurrent continuous queries. The system shows satisfactory scalability for concurrent path queries, and acceptable response times for concurrent range queries. Consequently, a general analysis and assessment of the algorithms suggest that our approach can be used for real-time services. Moreover, in some scenarios where the number of concurrent users becomes high, the performance can be increased by adopting a distributed data management

approach such as the one described in [Afyouni et al., 2013].

7. Conclusions

This paper introduces several algorithms for the management and processing of continuous location-dependent queries in indoor environments. Those algorithms are designed on top of a hierarchical data model that takes into account other contextual dimensions besides the location of the involved entities (e.g., time, user profiles that may imply restricted permissions to access certain areas, etc.). Two algorithms for continuous location-dependent queries over moving objects are introduced, implemented, and their performance evaluated. The former represents an incremental and hierarchical path search that can be executed at different levels of granularity, and applied on static and/or mobile data. The latter performs continuous range searches by applying a hierarchical network expansion mechanism and an incremental Euclidean restriction approach. These algorithms form the basis for an extensible query language grammar supporting continuous location-dependent searches.

The whole approach has been fully implemented as a pure database solution based on the PostgreSQL DBMS. Experiments have been conducted to investigate the scalability and performance with respect to the intrinsic properties of the proposed solutions. Results show that our proposal achieves a satisfactory performance, and it is efficient enough to be used in a real scenario. Experimental results show a mean execution time of around 0.2 second for continuous path searches, even in cases where the distance is quite large for an indoor scenario, and reasonable response times for continuous range searches. Furthermore, the whole system has been tested for scalability with respect to the number of concurrent users issuing a continuous query. The results show that the system is fairly scalable and adapted to a multi-user environment.

Future work is oriented towards: (1) introducing a policy based on lazy updates to reduce extra computations based on location granules; (2) integrating an extended context model that incorporates users' activities as well as content generated by other social entities into the location-dependent query processing; and (3) potentially supporting a Data Stream Management System (DSMS), such as TelegraphCQ [Chandrasekaran et al., 2003], for the processing of continuous queries over spatial data streams.

Acknowledgements

This research was partially supported by a Short Term Scientific Mission performed by the first author at the University of Zaragoza and funded by the COST Action IC0903 on "Knowledge Discovery from Moving Objects" (MOVE project). We would also like to acknowledge the support of the CICYT project TIN2010-21387-C02-02 and DGA-FSE.

References

Afyouni, I., Ilarri, S., Ray, C., Claramunt, C.. Context-aware modelling of continuous location-dependent queries in indoor environments. Journal of Ambient Intelligence and Smart Environments 2013;5(1):65–88.

Afyouni, I., Ray, C., Claramunt, C.. A fine-grained context-dependent model for indoor spaces. In: Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness. ACM; 2010. p. 33–38.

Afyouni, I., Ray, C., Claramunt, C.. Spatial models for indoor and context-aware navigation systems: A survey. Journal of Spatial Information Science 2012a;4(1):85–123.

Afyouni, I., Ray, C., Ilarri, S., Claramunt, C.. Algorithms for continuous location-dependent and context-aware queries in indoor environments. In: Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. ACM; 2012b. p. 329–338.

Baldauf, M., Dustdar, S., Rosenberg, F.. A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing 2007;2(4):263–277.

Becker, C., Durr, F.. On location models for ubiquitous computing. Personal and Ubiquitous Computing 2005;9(1):20–31.

Botea, A., Muller, M., Schaeffer, J.. Near optimal hierarchical path-finding. Journal of Game Development 2004;1(1):7–28.

Brinkhoff, T.. A framework for generating network-based moving objects. GeoInformatica 2002;6(2):153–180.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., Shah, M.. TelegraphCQ: Continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. ACM; 2003. p. 668–668.

Deng, K., Zhou, X., Shen, H., Sadiq, S., Li, X.. Instance optimal query processing in spatial networks. The VLDB Journal 2009;18(3):675–693.

Gu, Y., Lo, A., Niemegeers, I.. A survey of indoor positioning systems for wireless personal networks. IEEE Communications Surveys & Tutorials 2009;11(1):13–32.
Ilarri, S., Bobed, C., Mena, E.. An approach to process continuous location-dependent queries on moving objects with support for location granules. Journal of Systems and Software 2011;84(8):1327–1350.
Ilarri, S., Mena, E., Illarramendi, A.. Location-dependent queries in mobile contexts: Distributed processing using mobile agents. IEEE Transactions on Mobile Computing 2006;5(8):1029–1043.
Ilarri, S., Mena, E., Illarramendi, A.. Location-dependent query processing: Where we are and where we are heading. ACM Computing Surveys 2010;42(3):1–73.
IndoorAtlas, L.. Ambient magnetic field-based indoor location technology: Bringing the compass to the next level. IndoorAtlas Ltd.; 2012.
Jensen, C., Lu, H., Yang, B.. Graph model based indoor tracking. In: Proceedings of the 10th IEEE International Conference on Mobile Data Management (MDM). IEEE; 2009. p. 122–131.
Lee, C., Wu, Y., Chen, A.. Continuous evaluation of fastest path queries on road networks. In: Proceedings of the 10th International Conference on Advances in Spatial and Temporal Databases. Springer; 2007. p. 20–37.
Lee, D., Zhu, M., Hu, H.. When location-based services meet databases. Mobile Information Systems 2005;1(2):81–90.
Liu, H., Darabi, H., Banerjee, P., Liu, J.. Survey of wireless indoor positioning techniques and systems. IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews 2007;37(6):1067–1080.
Matthew, N., Stones, R.. Beginning databases with postgreSQL: From novice to professional. Apress, 2005.
Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.. Query processing in spatial network databases. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB). VLDB Endowment; 2003. p. 802–813.
Raubal, M.. Human wayfinding in unfamiliar buildings: A simulation with a cognizing agent. Cognitive Processing 2001;2(3):363–388.
Ray, C., Comblet, F., Bonnin, J.M., Le Roux, Y.M.. Wireless and information technologies supporting intelligent location-based services. In Book: Wireless Technologies in Intelligent Transportation Systems, Chapter 9, M.-T. Zhou, Y. Zhang, L.T. Yang (eds.); Nova Science Publishers. p. 225–265.
Schiller, J., Voisard, A.. Location-Based Services. San Francisco, CA, USA: Morgan Kaufmann, 2004.
Sun, X., Yeoh, W., Koenig, S.. Efficient incremental search for moving target search. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence (IJCA). Morgan Kaufmann; 2009. p. 615–620.
Sun, X., Yeoh, W., Koenig, S.. Generalized Fringe-Retrieving A*: Faster moving target search on state lattices. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. International Foundation for Autonomous Agents and Multiagent Systems; AAMAS '10; 2010a. p. 1081–1088.
Sun, X., Yeoh, W., Koenig, S.. Moving target D* lite. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. International Foundation for Autonomous Agents and Multiagent Systems; 2010b. p. 67–74.
Terry, D., Goldberg, D., Nichols, D., Oki, B.. Continuous queries over append-only databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM; 1992. p. 321–330.
Wang, H., Zimmermann, R.. Processing of continuous location-based range queries on moving objects in road networks. IEEE Transactions on Knowledge and Data Engineering 2011;23(7):1065–1078.
Wolfson, O., Sistla, A., Chamberlain, S., Yesha, Y.. Updating and querying databases that track mobile units. Distributed and parallel databases 1999;7(3):257–387.
Wu, K., Chen, S., Yu, P.. Incremental processing of continual range queries over moving objects. IEEE Transactions on Knowledge and Data Engineering 2006;18(11):1560–1575.
Xu, J., Guting, R.. Infrastructures for research on multimodal moving objects. In: Proceedings of the 12th IEEE International Conference on Mobile Data Management (MDM). IEEE; 2011. p. 329–332.
Yang, B., Lu, H., Jensen, C.. Scalable continuous range monitoring of moving objects in symbolic indoor space. In: Proceeding of the 18th Conference on Information and Knowledge Management (ICIKM). ACM; 2009. p. 671–680.
Yang, B., Lu, H., Jensen, C.. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In: Proceedings of the 13th International Conference on Extending Database Technology (EDBT). ACM; 2010. p. 335–346.
Yu, S., Spaccapietra, S.. A knowledge infrastructure for intelligent query answering in location-based services. Geoinformatica 2010;14(3):379–404.
Yuan, W., Schneider, M.. Supporting continuous range queries in indoor space. In: Proceedings of the 11th International Conference on Mobile Data Management (MDM). IEEE; 2010. p. 209–214.
Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.. Location-based spatial queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM; 2003. p. 443–454.