Interactive Route Search in the Presence of Order Constraints

Yaron Kanza[*]
Faculty of Computer Science
Technion
Haifa 32000, ISRAEL
kanza@cs.technion.ac.il

Roy Levin[*]
Faculty of Computer Science
Technion
Haifa 32000, ISRAEL
royl@cs.technion.ac.il

Eliyahu Safra
safraeli@gmail.com

Yehoshua Sagiv[†]
Dept. of Computer Science
The Hebrew University
Jerusalem 91904, ISRAEL
sagiv@cs.huji.ac.il

ABSTRACT

A *route search* is an enhancement of an ordinary geographic search. Instead of merely returning a set of entities, the result is a route that goes via entities that are relevant to the search. The input to the problem consists of several search queries, and each query defines a type of geographical entities. When visited, some of the entities succeed in satisfying the user while others fail to do so; however, only the probability of success is known prior to arrival. The main task is to find a route that visits at least one satisfying entity of each type. In an *interactive search*, the route is computed in steps. In each step, only the next entity of the route is given to the user, and after visiting that entity, the user provides a feedback specifying whether the entity satisfies her.

This paper investigates interactive route search in the presence of order constraints that specify that some types of entities should be visited before others. We present heuristic algorithms for interactive route search for two cases, depending on whether the constraints define a complete order or a partial one. The main challenge is to utilize the feedback in order to compute a route that is shorter and has a higher degree of success, compared to routes that are computed non-interactively. We also discuss how to compare the results of the algorithms and introduce suitable measures for doing so. Experiments on real-world data illustrate the efficiency and effectiveness of our algorithms.

[*]The work of these authors was supported by the German-Israeli Foundation for Scientific Research & Development (Grant 2165-1738.6/2007).
[†]The work of this author was supported by the German-Israeli Foundation for Scientific Research & Development (Grant 973-150.6/2007).

Keywords

Geographic information system, route, path, search, probabilistic data, heuristic algorithms, interactive algorithms

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Frequently, a user actually wants to visit the entities found in a geographic search that she performs. This requires providing the user not only with entities that satisfy the search conditions, but also with a route that leads to these entities. The need for a route is intensified when several geographical searches are joined to render a combined search task. Forming a route in this case is a rather difficult task due to the need to decide which object should be taken from the result of each search, how to order these objects, and whether to take more than one object from each result. The next example illustrates this.

Example 1.1. *Suppose that on her way from the office to a business meeting, Alice needs to fill up her gas tank, draw cash from an ATM, buy a new battery for her laptop, and go by a place where there is an Internet connection, in order to check her email. Suppose that Alice can conduct a simple geographic search using her cellular phone or car navigation system. She will be able to locate some nearby ATMs, some close gas stations, coffee shops that provide Internet connection, and electronics stores. However, combining the results of these searches into an efficient route that eventually leads to the location of the meeting can be a hard task.*

A route search, as the one in the example above, is a task of computing a route that starts at a given location, which is usually the location of the user, ends at a specified location and goes via geographic entities of certain types.

The geographic entities are considered as the *user needs* and are specified by search queries.

One of the difficulties when computing a route is dealing with *uncertainty*, namely, entities that are returned by the search queries, but actually do not satisfy the user needs. In the example above, Alice may find an electronics store that is located near the place of the meeting. Yet, upon arrival at the store, she may discover that the store does not have the specific battery she needs. She may also find out that on her way, she passed close to some other electronics stores; however, now there is no such store near her and she needs to lengthen her travel or go back to a place she already visited.

For dealing with uncertainty caused by entities that do not satisfy the user needs, we use a probabilistic model. In this type of model, each object has a *probability of success* which is the probability that the entity will satisfy the user needs. The probabilities can be generated from collected statistics. Such statistics, for instance, may show that most of the people who search for an ATM are satisfied with the result of the search. In this case, ATMs will receive high probabilities. The statistics may show that in only 80 percent of the cases, people who search for a restaurant eventually order food in some restaurant that has been discovered in the search. In such a case, a restaurant entity will receive a probability of 0.8. User profiling can be used for adjusting the probabilities to specific users. For instance, the economic status of a user may increase the probability of some restaurants and decrease the probability of others.

When computing a route over probabilistic data, there are two conflicting goals. One goal is that the route will be as short as possible. The other goal is that the route will go via objects that have the highest possible probabilities of satisfying the user needs. Semantics and algorithms for route search over probabilistic data were studied in a previous work [5]. This paper deals with *interactive route search.*

In an interactive route search, initially the user poses a route-search query; however, instead of providing to the user just one complete and unchanging route, the system creates the route gradually while interacting with the user. In each step, the system provides the next geographical entity on the route. The user goes to the entity and provides to the system feedback on whether the entity has satisfied her. The feedback is used when computing the rest of the route. Note that in this approach, the system can also present to the user a complete planned route, and modify the presented route whenever a feedback that changes the plan is received.

Example 1.2. *Consider the search task of Example 1.1. Suppose that the first entity Alice receives is a nearby Internet Cafe. Alice will go to the place and will provide a feedback to the system on whether she has been able to read her email. If the answer is positive, the rest of the route does not need to visit an Internet Cafe. If the answer is negative, the computation of the route continues and is required to satisfy the need for going by a place that provides Internet connection.*

For probabilistic datasets, computing routes iteratively can produce shorter routes than non-interactive evaluation. For instance, if a route goes via an entity of type T and the entity satisfies the user, there is no need to go by other entities of T. In the non-interactive approach, for comparison, it may be required to plan a route that goes via several entities of type T so that if one fails, another one will succeed. Thus, the iterative approach can shorten the length of the produced route.

Computing a route as short as possible, over a probabilistic dataset, is a difficult task. In the non-interactive case, the problem is NP-hard [5]. The interactive case is difficult as well for the following reason. For each entity it is required to consider the consequences of a success in satisfying the user and a failure to do so. Thus, although a single object is chosen in each step, the choice can be affected by an exponential number of scenarios.

Order constraints are used for specifying the order by which some types of entities should be visited. For instance, in the scenario of Example 1.1, Alice may need to visit an ATM and an electronics store before going to an Internet Cafe. The order constraints may define a *complete order* that specifies for each pair of types which one should be visited first. It may also define a *partial order* that specifies the visit order for some pairs of types, but does not specify it for the others.

In the presence of order constraints, the route-search algorithms need to guarantee that the objects are visited in an order that satisfies the constraints. Thus, the constraints are an additional factor that makes it harder to devise algorithms for route search. The case of a partial order is more difficult to handle, because one has to consider all the complete orders that are consistent with the given partial order. In the case of a complete order, there is just one order to consider and that makes the problem conceptually and computationally easier.

The main contributions of this paper are in formally defining the problem of interactive route search in the presence of order constraints, presenting interactive algorithms for route-search queries in the presence of order constraints, and comparing the efficiency and effectiveness of the algorithms experimentally and analytically.

2. RELATED WORK

Earlier works on route-search queries dealt either with non-probabilistic datasets (e.g., [1, 3, 7, 8, 9]), or with the non-interactive version of the problem (e.g., [5, 6]). Those results cannot be used to solve the interactive route-search problem that we address in this paper, since they are not designed to take user feedback into account.

Recently, interactive route search over probabilistic data has been introduced and investigated in [4]. There is a fundamental difference between [4] and this paper. The algorithms presented in [4] do not deal with order constraints and cannot be easily modified to do so efficiently and effectively. The reason for this is that when examining potential routes, it is required to consider only routes that satisfy the order constraints. However, doing that is intricate, because considering an exponential number of routes and discarding routes that do not satisfy the order constraints is inefficient. Computing a route regardless of the constraints and skipping entities, if visiting them violates some order constraint, is not effective. Hence, new algorithms should be devised for route search in the presence of order constraints.

Moreover, we take advantage of the fact that the query size is small. That is, the user has to visit a small number of destinations; however, the number of objects that could potentially satisfy the user is large. Thus, we assume that the query size is bounded, and we develop algorithms that perform a comprehensive global search, as opposed to the greedy local-search algorithms of [4]. Since the algorithms of this paper are more exhaustive than those of [4], they are more effective, even for queries without order constraints.

3. PROBABILISTIC ROUTE SEARCH

In this section, we present our framework, we formally define the concept of *interactive route search*, and we explain how order constraints affect route-search queries.

3.1 Geo-spatial Datasets

A *geo-spatial dataset* consists of a collection O of geo-spatial objects and a graph G of a road network that connects the objects. Each object represents a real-world geographical entity and its location is the same as that of the entity. In the sequel, "object" and "entity" are synonyms, although technically an object is a representation of a real-world entity.

Each edge in the graph G represents a segment of a real-world road and it has a length. The length of an edge is the length of the road segment it represents. That is, an edge with length ℓ between two objects o_1 and o_2 represents a real-world road with length ℓ connecting o_1 and o_2. We use $length(o_1, o_2)$ to denote the length of this edge.

A path in G from node o_1 to node o_2 is a sequence of nodes $o_1, o_2, \ldots, o_m$, such that every two adjacent nodes o_i and o_{i+1} are connected by an edge of G. The length of the path is the sum of the lengths of its edges, namely, $\Sigma_{i=1}^{m-1} length(o_i, o_{i+1})$. The *distance* between two objects o and o' is the length of the shortest path that connects them. We denote this distance by $dist(o, o')$.

3.2 Search Queries

Users employ *search queries* to specify the entities that they would like to visit. In this paper, we assume that a search query is just a set of keywords, but other alternatives are also possible. The result of a search query is represented as a *probabilistic dataset*, namely, each object is assigned a value $0 \leq p \leq 1$, called *probability of success* (or *probability*, for short). The probability of an object o specifies what is the likelihood that o represents an entity that actually satisfies the user's needs. The task of assigning probabilities to objects is beyond the scope of this paper.

3.3 Route-Search Queries

Route-search queries are generated by combining several search queries that specify different types of entities through which the route should go. We use Q (typically with a subscript) to denote a search query. We denote by $\mathcal{Q}$ a collection of several search queries that together constitute one component of a route-search query, as we explain later.

An *order constraint* on a route-search query $\mathcal{Q}$ is a pair (Q_1, Q_2), where Q_1 and Q_2 are distinct queries of $\mathcal{Q}$. Intuitively, this pair specifies that the user must visit a satisfying entity of the answer to Q_1 prior to arriving at an entity of the answer to Q_2. Users can add order constraints to a route-search query by specifying a set of such pairs.

Let C be a set of order constraints over $\mathcal{Q}$. The *precedence graph*, denoted by G_C, is a directed graph whose nodes are the search queries of $\mathcal{Q}$ and whose directed edges are the pairs of C. When there is a path in G_C from some query Q_1 to a query Q_2, we say that Q_1 *precedes* Q_2 and we denote this by $Q_1 \prec Q_2$. We say that C is a *valid* set of constraints if G_C is acyclic. It is easy to see that when there is a cycle in G_C, it is impossible to satisfy all the order constraints, because there are two queries such that each one should precede the other.

We say that G_C defines a *complete order* over $\mathcal{Q}$ if it contains a Hamiltonian path, that is, a directed path that goes via all the elements of $\mathcal{Q}$. Otherwise, we say that G_C defines a *partial order*.

In a *route-search query*, the user specifies a *start location* s, a *target location* t, a set $\mathcal{Q}$ of search queries, and a valid set C of order constraints. Hence, we represent a route-search query as a 4-tuple $R = (s, t, \mathcal{Q}, C)$.

EXAMPLE 3.1. *Consider again Example 1.1. A suitable route-search query for Alice should include* (1) *the location s of her office,* (2) *the location t where the meeting should be held, and* (3) *the following four search queries:* $Q_1 = \{\texttt{gas station}\}$, $Q_2 = \{\texttt{ATM}\}$, $Q_3 = \{\texttt{laptop battery}\}$, *and* $Q_4 = \{\texttt{Internet Cafe}\}$. *The order constraints* (Q_2, Q_4) *and* (Q_3, Q_4) *specify that Alice should visit an ATM and an electronics store before going to the Internet Cafe. Note that there is no order constraint that involves* Q_1*, which means that a gas station can be located anywhere on the route.*

Consider a route-search query $R = (s, t, \mathcal{Q}, C)$, where $\mathcal{Q}$ is the set $\{Q_1, \ldots, Q_m\}$ of search queries. The *answer set* (or *result*) of Q_i, denoted by A_i, comprises the objects of the database that are relevant to Q_i. For simplicity, we assume that the answer sets $A_1, \ldots, A_m$ are pairwise disjoint.[1] Namely, distinct search queries of $\mathcal{Q}$ refer to different types of objects. For example, one search query is about hotels, another is concerning restaurants, etc. A *pre-answer* to R is a sequence $s, o_1, \ldots, o_k, t$ that starts at s, ends at t and goes via objects of the results $A_1, \ldots, A_m$, such that every A_i has at least one object in the sequence. The objects are visited in an order that conforms to the constraints of C. That is, for all o_{i_1} and o_{i_2}, where $i_1 < i_2$, it holds that if o_{i_1} belongs to A_{j_1} and o_{i_2} belongs to A_{j_2}, then Q_{j_2} does not precede Q_{j_1} (i.e., in G_C there is no path from Q_{j_2} to Q_{j_1}, so $Q_{j_2} \not\prec Q_{j_1}$).

The *length* of the route is the sum of the distances between consecutive objects, that is,

$$dist(s, o_1) + \Sigma_{i=1}^{k-1} dist(o_i, o_{i+1}) + dist(o_k, t) .$$

3.4 Interactive Search

Answering route-search queries is traditionally done by computing a complete route, from s to t, that has a high probability-of-success and a short length [5]. An interactive search is different from the traditional approach in the following aspect. After visiting an entity, the user provides a feedback on whether the entity actually satisfies the corresponding search query, and only then does the system determine the next entity to be visited. In other words, instead of computing a complete route in advance, the route is computed incrementally. At each step, the system provides to the user a single object, which is the next object on the route. After visiting the geographical entity that corresponds to the object, the user sends to the system information on whether the entity satisfies her needs, and this feedback affects the computation of the following objects on the route. Alternatively, the system may give to the user a complete route (that visits relevant objects of the search

[1]When an object appears in k different answer sets, we can consider it as k distinct objects having a shared location.

queries that still have to be satisfied). The system may change this route when the feedback warrants doing so.

The computation of the route is influenced by the order constraints. When the user visits an entity that meets her needs, the corresponding search query is deemed satisfied. In each step, the user can visit an entity only if the corresponding object o is an answer to a search query Q_i, such that all the queries that precede Q_i have already been satisfied. When all the queries have been satisfied, the user goes to the target location t and the search ends. Recall that when there are m search queries in $\mathcal{Q}$, there is a need to visit exactly m entities that satisfy the user.

Note that if all the objects of some answer set A_j have already been visited, and none has satisfied the user, then R cannot be satisfied. In this case, a failure message should be sent to the user and a new search should be initiated.

Our goal is to develop algorithms for interactive route search that compute routes that are as short as possible.

4. ALGORITHMS

In this section, we describe interactive algorithms for route search. Each algorithm has two versions: one is for queries whose constraints define a complete order, and the second version is for queries whose constraints define a partial order. All the algorithms operate over the objects in the answer sets $A_1, \ldots, A_m$ of the search queries of $\mathcal{Q}$, and they compute a route by iteratively increasing a partial sequence σ. Initially, the partial sequence comprises only the start location, namely, $\sigma = s$. On each iteration, the algorithms provide to the user the next object o_k to be visited; thus, o_k is added at the end of σ. When arriving at o_k, the user provides a feedback regarding whether o_k actually satisfies the corresponding search query (i.e., the query Q_i, such that $o_k \in A_i$). The feedback determines whether the objects of A_i are still relevant to the search and whether Q_i is satisfied.

For each object o in the sequence σ, we denote by $o\text{-}sat(o)$ the feedback received for o. When this feedback is `true`, it means that the object satisfies the corresponding search query. Otherwise (i.e., in the case of a `false` feedback), the object does not satisfy the query.

On each iteration, an object is chosen from the answers to the queries that have not yet been satisfied. Next, we formally define the set from which the object is chosen. Consider a route-search query $R = (s, t, \mathcal{Q}, C)$, where $\mathcal{Q} = Q_1, \ldots, Q_m$. Let $\sigma = s, o_1, \ldots, o_k$ be the partial sequence computed so far. The *unsatisfied queries* of R are all the queries Q_i, such that σ has no object that satisfies Q_i. In other words, we use $q\text{-}unsat_R(\sigma)$ to denote the set of these queries and define $q\text{-}unsat_R(\sigma) = \{Q_i \mid Q_i \in \mathcal{Q} \text{ and } \neg\exists o(o \in \sigma \wedge o \in A_i \wedge o\text{-}sat(o))\}$, where A_i is the answer set for Q_i.

In each iteration, the sequence σ is extended by providing to the user the next object of the route. The added object is chosen from a set of *candidate objects*. This set, denoted $candidates_R(\sigma)$, consists of all objects o, such that o has not yet been visited and its addition to σ complies with the order constraints. When computing the set of candidate objects, we use the precedence graph G_C that is generated from the order constraints C. Let G_{unsat} be the induced subgraph of G_C w.r.t. the unsatisfied queries of $q\text{-}unsat_R(\sigma)$. That is, G_{unsat} is obtained from G_C by removing all the satisfied queries and their incident edges. Let $\mathcal{Q}_0$ be the set of nodes of G_{unsat} with no incoming edges (i.e., queries that have no preceding query in G_{unsat}). Then, o is a candidate object if o does not appear in σ and is an answer to some query of $\mathcal{Q}_0$. In Appendix G, we show how to use an index to reduce the number of candidate objects, for efficiency reasons. When $\mathcal{Q}_0$ is the empty set, all the queries have been satisfied (i.e., $q\text{-}unsat_R(\sigma) = \emptyset$) and the route must continue to the end location t. If there is no candidate object and $\mathcal{Q}_0$ is not empty, then the route-search query R cannot be satisfied. Note that when C defines a complete order on $\mathcal{Q}$, then in each iteration (except for the last one where $\mathcal{Q}_0$ is empty), $\mathcal{Q}_0$ contains exactly one query, and thus all the candidate objects are answers to the same query.

4.1 Greedy Algorithms

The *naive greedy heuristic* is a simple method that serves as our baseline; namely, more elaborate algorithms will be compared to it. In each iteration, this heuristic chooses the candidate object that is closest to the current location l.

The naive greedy heuristic is simple and efficient. However, it suffers from the drawback of ignoring the location of the target t. Consequently, it may compute a route that drifts far away from t and is unnecessarily long. The *oriented greedy heuristic* is aimed at solving this problem by choosing the next object o' so that it will be near the current location as well as close to the straight line from s to t. Since the greedy algorithms are simple, and the greedy approach is not novel, we specify their details in the appendix.

4.2 Optimistic Approach

The main weakness of the greedy approach is that in each step it employs a local search, i.e., it chooses the next object o' without taking into account the length of the route from o' to t. A more thorough algorithm should consider in each step the overall length of a route that starts at the current location, passes through objects that satisfy the remaining search queries and ends at t. The *optimistic approach* does that by computing at each iteration a complete route with respect to the search queries that still have to be satisfied. We now describe how it works.

The algorithm computes the shortest pre-answer, that is, as short a route as possible from the start location to the end location via one object from each answer set $A_1, \ldots, A_m$. The user follows this route till an object fails to satisfy its corresponding search query. When that happens, the algorithm finds the set of yet unsatisfied queries $\mathcal{Q}_u \subseteq \mathcal{Q}$ and computes a new route, from the current location to t, that goes via one object of each $A_i \in \mathcal{A}_u$, where $\mathcal{A}_u$ are the answers to the unsatisfied queries $\mathcal{Q}_u$.

This approach is “optimistic” in the sense that at each step, the route is computed under the assumption that all the relevant objects satisfy their corresponding queries. If this assumption holds, the shortest pre-answer is indeed the optimal solution. Next, we explain in more detail the two versions of this approach—one for the case where the constraints define a complete order, and another for the case where the constraints define a partial order.

4.2.1 Optimistic Approach for Complete Orders

For route-search queries $R = (s, t, \mathcal{Q}, C)$ whose constraints define a complete order, we can efficiently compute the shortest pre-answer. Without loss of generality, suppose that the constraints define the order $Q_1, \ldots, Q_m$ over the queries of $\mathcal{Q}$ (i.e., objects of Q_1 should be visited first, then objects of Q_2, after that objects of Q_3, and so on). Consider the

```
Ordered Optimistic $((s, t, \mathcal{Q}, C), D)$
Input: Start location $s$, target location $t$, search queries
$Q_1, \ldots, Q_m$ ordered according to $C$, a dataset $D$
Output: A route that satisfies the search queries $Q_i$ in
the order of increasing $i$, based on feedback from the user
 1: $u\text{-}location \leftarrow s$
 2: for $i = 1$ to $m$ do
 3:   $found \leftarrow false$
 4:   while $A_i \neq \emptyset$ and not $found$ do
 5:     $o \leftarrow \operatorname{argmin}_{o \in A_i}(dist(u\text{-}location, o) + dist\text{-}t(o))$
 6:     provide $o$ to the user and receive a feedback
 7:     $u\text{-}location \leftarrow$ the location of $o$
 8:     if $o$ satisfies $Q_i$ then
 9:       $found \leftarrow true$
10:     else
11:       $A_i \leftarrow A_i - \{o\}$
12:   if not $found$ then
13:     return "the route cannot be completed"
14: provide the target destination $t$ to the user
```

Figure 1: Optimistic algorithm when C defines a complete order

answer sets $A_1, \ldots, A_m$ of R. For each $o \in A_i$ $(1 \leq i \leq m)$, we compute the minimal distance of a route that starts at o and for $j = i+1, \ldots, m$, passes through one object of each A_j in the order of increasing j, and finally arrives at t. We denote this minimal distance by $dist\text{-}t(o)$ and refer to it as the *distance-to-target* of o. An algorithm for computing the $dist\text{-}t(o)$ values is presented in Appendix C.

The optimistic algorithm for route-search queries $R = (s, t, \mathcal{Q}, C)$, where C defines a complete order, is shown in Figure 1. This algorithm computes a route that satisfies the search queries Q_i in the order of increasing i. In each iteration, it suffices to compute only the next object to be visited, rather than a whole route. Line 1 sets the current location to s. The loop of Line 2 iterates through the answer sets A_i in the order of increasing i. For each A_i, the loop of Line 4 iterates over objects of A_i until it finds one that satisfies Q_i. In Line 5, the algorithm picks the object o of A_i that appears on the shortest pre-answer (w.r.t. $Q_i, \ldots, Q_m$) from the current location to t. In Line 6, the user is informed to travel to o and provides her feedback. Line 7 sets the current location to that of o. The test of Line 8 checks whether o satisfies Q_i. If the test is positive, then Line 9 sets *found* to `true`, which means that the while loop of Line 4 terminates and the algorithm proceeds to the next iteration of Line 2. Otherwise (i.e., o does not satisfy Q_i), the object o is removed from A_i and another iteration of the loop of Line 4 is done. If A_i becomes empty before finding an object that satisfies Q_i, the algorithm terminates in Line 13 after notifying the user that the route cannot be completed. When the loop of Line 2 terminates (without reaching Line 13), the user is informed to travel to the target location.

4.2.2 *Optimistic Approach for Partial Orders*

In case of a complete order, computing the distance-to-target values is rather straightforward, because the shortest route from the current location to t is unique. In case of a partial order, the shortest route may vary depending on the types of objects that have already been visited. As an example, suppose that there is no order constraint that involves Q_i, i.e., an object of A_i may appear anywhere on the route. If the current location is an object $o \in A_j$, where $j \neq i$, then we should consider (at least) two distinct shortest routes from o to t; one of those routes visits an object of A_i while the other does not. In other words, the distance-to-target value of o depends on whether an object of A_i has already been visited or not. Thus, we should compute the distance-to-target value of o for each possible *history,* namely, each sequence of queries that have already been satisfied.

Formally, we first construct the set $\mathcal{O}_C$ of all the complete orders over $\mathcal{Q}$ that conform to the constraints of C. Next, consider an object $o \in A_i$. We have to compute for o a distance-to-target value for each sequence $Q_{i_1}, \ldots, Q_{i_f}$ of distinct search queries, such that $Q_{i_1}, \ldots, Q_{i_f}, Q_i$ is a prefix of some element of $\mathcal{O}_C$. We do it by considering every suffix $Q_{i_g}, \ldots, Q_{i_m}$, such that $Q_{i_1}, \ldots, Q_{i_f}, Q_i, Q_{i_g}, \ldots, Q_{i_m}$ is in $\mathcal{O}_C$. We compute the distance-to-target value of o w.r.t. the complete order $Q_{i_1}, \ldots, Q_{i_f}, Q_i, Q_{i_g}, \ldots, Q_{i_m}$ using the algorithm of Figure 2. The actual distance-to-target value of o w.r.t. the sequence $Q_{i_1}, \ldots, Q_{i_f}$ is the minimum over all the possible suffixes. This computation is based on the assumption that all the objects that correspond to the queries of a possible suffix $Q_{i_g}, \ldots, Q_{i_m}$ are available, namely, none of them has already been visited and failed. However, this is not necessarily true, because the partial order implied by the constraints of C may allow objects corresponding to some Q_j $(j \neq i)$ to be visited either before or after o. Therefore, the computed distance-to-target value is only an estimation. We elaborate on this in the Appendix E. In summary, we create for each object o an *estimated-distance table* (EDT) that maps sequences of search queries to distance-to-target values. Finally, observe that if two distinct sequences consist of exactly the same queries, then the same value is computed for both. Hence, the entries of an EDT are subsets of $\mathcal{Q}$ rather than sequences. The following example illustrates what are the entries of an EDT.

EXAMPLE 4.1. *Consider a route-search query where $\mathcal{Q} = \{Q_1, Q_2, Q_3, Q_4, Q_5\}$ and $C = \{Q_1 \prec Q_2,\ Q_2 \prec Q_3,\ Q_2 \prec Q_4,\ Q_3 \prec Q_5,\ Q_4 \prec Q_5\}$. There are two complete orders to consider: Q_1, Q_2, Q_3, Q_4, Q_5 and Q_1, Q_2, Q_4, Q_3, Q_5. Now, for an object o_2 in the result of Q_2, the EDT has a single entry, which maps the set $\{Q_1\}$ to the shortest distance among the following two routes: (1) the shortest pre-answer from o_2 to t with respect to the complete order Q_2, Q_3, Q_4, Q_5, and (2) the shortest pre-answer from o_2 to t with respect to the complete order Q_2, Q_4, Q_3, Q_5.*

For an object o_3 in the result of Q_3 there are two entries in the EDT, one is for the set $\{Q_1, Q_2\}$ and the other is for the set $\{Q_1, Q_2, Q_4\}$.

The optimistic approach starts the processing of a route-search query by constructing an EDT for every object. The route is computed in stages as follows. Let $\sigma = s, o_1, \ldots, o_k$ be the sequence of objects visited thus far (initially $\sigma = s$). We use $q\text{-}sat_R(\sigma)$ to denote the set of queries that have been satisfied by σ (i.e., $q\text{-}sat_R(\sigma) = \mathcal{Q} \setminus q\text{-}unsat_R(\sigma)$). For an object o that has an entry for $q\text{-}sat_R(\sigma)$ in its EDT, let $d_\sigma(o)$ be the value of that entry. The next object to be visited is the one that minimizes the sum $dist(o_k, o) + d_\sigma(o)$, among all objects o that have an entry for $q\text{-}sat_R(\sigma)$ in their EDT and are not already in σ.

4.3 The Effect of Probabilities on the Route

Thus far, we presented greedy algorithms and optimistic algorithms that consider only the distances between objects, but ignore the probabilities. One way to add to these algorithms the effect of the probabilities is by changing the distance function. For every two objects o_1 and o_2, we define the distance function $dist_p(o_1, o_2)$ to be $dist(o_1, o_2)/prob(o_2)$. We can now use $dist_p$ instead of $dist$. This increases the distance to objects with a low probability of success in a manner that is inversely proportional to the probability.

4.4 Minimizing the Expected Distance (MED)

The optimistic approach employs a best-case scenario. That is, the next object to be visited is the first on the shortest route that passes through one object of each answer set A_i, such that Q_i has not yet been satisfied. A more realistic approach is to use an average-case analysis. The main idea is to choose the next object based on the expected, rather than the shortest, distance that still remains to be traveled. To formalize this notion, let s be the current location and consider an object o. The following is a recursive definition of the expected distance to be covered, given that o is the next object to be visited. There are some expected distances ℓ_s and ℓ_f from o to the target location,[2] depending on whether o succeeds (i.e., satisfies its corresponding query) or fails, respectively. Thus, given that o is the next object, the expected distance from the current to the target location is the following sum.

$$dist(s, o) + prob(o) \cdot \ell_s + (1 - prob(o)) \cdot \ell_f \tag{1}$$

In the MED approach, the next object o to be visited is one that minimizes the above sum.

Computing the expected distance for an object o is not easy. First, there could be an exponential number of pre-answers that need to be considered. Second, we should avoid pre-answers that visit the same object more than once, which means that when constructing the pre-answers, we should keep the entire history (i.e., the visited objects) of each pre-answer. Doing so for an exponential number of pre-answers is impractical. Hence, we use heuristics that estimate the expected distance, rather than compute it precisely. Examples that illustrate the difference between MED and the other algorithms are provided in Appendix I.

4.4.1 MED for Complete Orders

In this section, we describe the version of MED for route-search queries with a complete order. The crux of the algorithm is an estimation of the expected distance, given that objects must be visited in the order dictated by the constraints. Actually, an expected distance is computed for each object o, assuming that the route starts at o and has to satisfy the search query Q that corresponds to o as well as all the search queries that succeed Q in the complete order. As explained later in this section, this estimation employs a heuristic that enforces a total order on the objects of the dataset D, thereby limiting the number of examined routes. The expected distances are stored in an array E. For each object o, the entry $E[o]$ estimates the expected distance of the shortest route that satisfies all the remaining search queries, starting with the one corresponding to o. After computing the array E, the rest of the MED algorithm is straightforward. In each step, the next object o is the one that minimizes the expected travel distance given by Equation 1 and $E[o]$ (the latter estimates the sum of the last two terms of Equation 1). The MED algorithm for complete orders is described in detail in Appendix D.

In the remainder of this section, we describe how to compute the estimation $E[o]$ for all objects $o \in D$. First, we define the order $\prec$ over the objects of D. When doing so, we assume that the queries $Q_1, \ldots, Q_m$ and their corresponding answers $A_1, \ldots, A_m$ are ordered according to $\prec$, i.e., if $i < j$ then $Q_i \prec Q_j$. Now, if $o_1 \in A_i$, $o_2 \in A_j$ and $i < j$, then naturally $o_1 \prec o_2$, because objects of A_i must be visited before objects of A_j. In order to define $\prec$ for objects of the same answer set A_i, we partition the straight line from s to t into $m + 1$ equal intervals. Let the sequence of points $p_0, p_1, \ldots, p_m, p_{m+1}$ describe this partition, where $p_0 = s$ and $p_{m+1} = t$. In other words, the intervals $[p_i, p_{i+1}]$ $(0 \leq i \leq m)$ cover the straight line from s to t, they are disjoint and have the same length. Objects of the same answer set A_i are ordered according to their distance from p_i. That is, $o_1 \prec o_2$ if o_1 and o_2 are both in A_i and $dist(o_1, p_i) < dist(o_2, p_i)$. In case of a tie (i.e., $dist(o_1, p_i) = dist(o_2, p_i)$), the order between o_1 and o_2 is defined arbitrarily.

The rationale for the above definition is to prefer objects that are closer to the line from s to t and, in particular, objects whose distance from s is linearly proportional to their position on a possible route. In other words, the goal is to choose the next object so that it will be in the direction toward to t, but not too close to t, in order to avoid routes that unnecessarily go back and forth.

When estimating the expected length of a route, we should take into account the possibility that some search queries are not satisfied by any object. To do it properly, we define for each answer set A_i a *penalty* that amounts to the length of a route that goes through all the objects of A_i (which must be done when Q_i cannot be satisfied). That is, $penalty(A_i) = \Sigma_{j=1}^{k-1} dist(o_j^i, o_{j+1}^i)$, where the route $o_1^i, \ldots, o_k^i$ passes exactly through all the objects of A_i from the smallest to the largest, according to the order $\prec$. (In Appendix D, we explain why the penalty has been defined this way.)

Recall that $E[o]$ denotes our estimation of the expected length of the shortest route from an object o of A_i to t, such that the search queries $Q_i, \ldots, Q_m$ are satisfied. $E[o]$ is given by

$$\begin{aligned} E[o] = prob(o) \cdot (dist(o, o_s) + E[o_s]) + \\ + (1 - prob(o)) \cdot (dist(o, o_f) + E[o_f]) \end{aligned} \tag{2}$$

where o_s and o_f are defined as follows. If o succeeds, then an object of A_{i+1} should be visited next. Therefore, we choose o_s from the objects of A_{i+1} so that the sum $dist(o, o_s) + E[o_s]$ is minimized, except that o_s is t if $i = m$ (note that by definition, $E[t] = 0$). If o fails, then another object of A_i should be visited next. To avoid an exponential computation, we choose o_f from the objects of A_i that are larger than o according to $\prec$. In particular, o_f is picked out so that the sum $dist(o, o_f) + E[o_f]$ is minimized; however, if o is the last object of A_i (according to $\prec$), then we replace the sum $dist(o, o_f) + E[o_f]$ with $penalty(A_i)$, because none of the visited objects of A_i satisfies Q_i.

[2]More precisely, the user travels until she either arrives at t or discovers that one of her search queries cannot be satisfied. The expected distance is computed by considering the probabilities and lengths of all the routes she may travel.

The algorithm *ComputeExpLen* computes all the entries of E by means of Equation (2). To make the process well defined, the algorithm iterates over the objects of D from the largest to the smallest, according to $\prec$. The pseudo-code is given in Figure 4 of Appendix D.

4.4.2 MED for Partial Orders

The adaptation of MED to partial orders is obtained in a way similar to how it was done in the case of the optimistic approach. An estimation of the expected distance has to be computed for every pair (o, S), such that o is an object in the answer set of some query and S is a subset of $\mathcal{Q}$ that represents a possible history, namely, the search queries of S have already been satisfied before arriving at o.

Formally, let S be a subset of $\mathcal{Q}$ and Σ be a sequence $Q_{i_g}, \ldots, Q_{i_m}$ of distinct queries. Recall that $\mathcal{O}_C$ is the set of all the complete orders implied by the constraints of C. We can view each element of $\mathcal{O}_C$ just as a sequence. We say that Σ is *consistent* with S if the following holds. (1) Σ is a suffix of some element of $\mathcal{O}_C$; (2) No search query appears in both S and Σ; and (3) Every search query of $\mathcal{Q}$ appears in either S or Σ. We say that Σ is an *i-sequence* if Q_i is the first query in Σ.

Consider an object $o \in A_i$. The array E (of estimations) of expected distances has an entry for every pair (o, S), such that $S \subseteq \mathcal{Q}$ and there is some i-sequence Σ that is consistent with S. The value $E[o, S]$ is computed as follows.

Let Σ be an i-sequence that is consistent with S. We apply the algorithm *ComputeExpLen* of Figure 4 w.r.t. the complete order Σ (while ignoring objects corresponding to search queries that do not appear in Σ). We do it for every i-sequence that is consistent with S. The minimum value computed for o, over all the i-sequences that are consistent with S, is assigned to $E[o, S]$.

The above description of how to compute $E[o, S]$ is not the most efficient way of doing it. In fact, it suffices to apply the algorithm *ComputeExpLen* of Figure 4 once for each complete order of $\mathcal{O}_C$. (Recall that $\mathcal{O}_C$ is the set of complete orders implied by the constraints of C.) If we do it in this way, then we actually compute values of the form $E[o, \Gamma]$, where $\Gamma \in \mathcal{O}_C$. Let $\Gamma{<}o{>}$ denote the suffix of Γ that starts at the Q_i corresponding to o. $E[o, S]$ is the minimum over all $E[o, \Gamma]$, such that $\Gamma{<}o{>}$ is consistent with S.

More specifically, for each object o, we need to divide all the $E[o, \Gamma]$ into subsets, such that in each one all the Γ have the same search queries appearing before Q_i. Thus, each subset corresponds to one $E[o, S]$, where S is the set of search queries that appear before Q_i in all the Γ of the subset. $E[o, S]$ is assigned the minimum value in its corresponding subset.

As earlier, σ denotes the route traveled thus far, and $q\text{-}sat_R(\sigma)$ is the set of queries that have already been satisfied. The algorithm MED for partial orders is similar to MED for complete orders (given in Figure 3 of Appendix D). The main difference is changing Line 10 of Figure 3 so that the next object to be visited is the one that minimizes the sum $dist(curr, o) + E[o, q\text{-}sat_R(\sigma)]$ over all objects o, such that $E[o, q\text{-}sat_R(\sigma)]$ is defined, o has not yet been visited and its corresponding search query still has to be satisfied. If there is no such o, then the algorithm has failed to find a route. As usual, if the route σ has satisfied all the search queries, then the user should travel to the target location t.

5. EXPERIMENTS

In order to examine the effectiveness and efficiency of our methods, we tested them over real-world data in a variety of cases. We conducted many experiments and we present here only the results of typical cases. In our experiments we used real-world data. We simulated the satisfaction of objects according to their probabilities and we tested route-search queries R where the number of search queries in $\mathcal{Q}$ is between three to seven. See details in Appendix H.

5.1 Effectiveness

We conducted a series of experiments to examine the effectiveness of our algorithms. We tested the effect of different parameters on each algorithm. In the experiments we compared MED, Optimistic, Oriented Greedy and Naive Greedy. For Optimistic and Oriented Greedy, we experimented with the version that is affected by the probabilities, i.e., the version that uses $dist_p$ instead of $dist$. In this section, we denote the Optimistic by wOpt, the weighted oriented Greedy by wGre, and the Naive Greedy by bGre.

We conducted experiments using many different queries, as well as many different start and end points. However, we did not find any significant impact of setting different start/end locations, nor of using different search queries, on the comparison between the different algorithms. Thus, these parameters are not specified in the experiments below.

Comparing the algorithms one to the other only provides a relative indication of their effectiveness. For a non-relative comparison, we included in our experiments an algorithm we call *Perfect*. Perfect computes the shortest route while having the satisfaction conditions of all the objects before the first step. Since Perfect has information that no interactive algorithm has, the route computed by Perfect is the best that any interactive algorithm could hopefully compute. Obviously, in actual scenarios, such an algorithm does not exist; however, in our experiments, we had all the information on the objects, and hence, we were able to use it. We compare the results of our algorithms to the results of Perfect, in order to show that our algorithms are effective in general and not just relatively.

Figures 5, 6 and 7 show the results of tests that examine the effect of order constraints on the effectiveness of the algorithms.[3] In these graphs, the x-axis shows lengths. For each length ℓ, the y-axis presents the percentage of routes that were created interactively and had a length of at most ℓ. The percentage was achieved by running each route-search query 100 times, while simulating interaction with the user. When comparing two interactive algorithms on such a graph, the better algorithm is the one whose curve is higher because the routes it produces are expected to be shorter.

In the experiments whose results are presented in Figure 5, Figure 6 and Figure 7, probabilities where normally distributed[4] with mean 0.7 and standard deviation 0.1. The route-search queries in this experiment comprise five search queries, i.e., need to go via objects of five types.

Figure 5 shows the results of the algorithms for the case where there are no order constraints. There are 120 complete orders in this case. Figure 6 shows the results for the case where there is a partial order. There are 20 complete

[3]The figures of this section appear in the appendix.

[4]The actual distribution is close to normal since we do not allow objects to receive a probability lower than zero or greater than one.

orders in this case. The case where the constraints define a complete order is presented in Figure 7.

The results show that MED outperforms the other algorithms in almost all of the cases. Optimistic (wOpt) is almost as good as MED, and both of them are almost as good as Perfect, which shows that they are indeed effective. The Greedy algorithms bGre and wGre are less effective than MED and wOpt.

Figure 8 presents the results of comparing the algorithms to Perfect. For each algorithm, it shows the average difference between the length of the route computed by the algorithm and the length of the route computed by Perfect. The results are shown for the cases where the means of the probabilities are 0.3, 0.6 and 0.9, respectively. Not surprisingly, the results of the algorithms are closer to Perfect when probabilities are high than when probabilities are low. This experiment also shows that MED is the most effective in all cases. Optimistic is effective when the probabilities are high, but it is not effective when the probabilities are low. This is because it applies an "optimistic" assumption and when the probabilities are low, this assumption is incorrect. The greedy approach is relatively good when the probabilities are low, because in this case, most of the visited objects fail to satisfy the user, so not planning and going to the nearest object is a good strategy for this case.

Figures 9, 10 and 11 show the results of the different algorithms for a search over three datasets, where the probabilities are normally distributed with means 0.3, 0.6 and 0.9, respectively. In this experiment, the route-search query comprised three search queries, thus, the objects are partitioned into three categories. This experiment provides an additional affirmation to the effectiveness of MED.

In Appendix E, we presented the problem of phantom objects and claimed that a possible solution is to recalculate the estimation of the minimal distance after every negative feedback. We denote by rMED the algorithm Recalculate MED and by rwOpt the algorithm Recalculate Optimistic. Our experiments showed that recalculation has almost no effect on the effectiveness of the algorithms. The results of these experiments are presented in Figure 12. The test shows this by comparing the results of MED, rMED, wOpt and rwOpt to the results of Perfect. It is done over datasets in which the probability is normally distributed with means 0.3, 0.6 and 0.9, respectively. Each column is the average over three different start and target locations, and for 100 different interactive runs. It can be seen that there is almost no difference between MED and rMED. Similarly, there is almost no difference between wOpt and rwOpt.

5.2 Efficiency

We analyzed the complexity of our algorithms (see details in Appendix F) and tested them experimentally. We conducted our tests on a standard computer with Intel Core 2 Duo 2.26 GHz CPU and 2 GB of RAM. All the algorithms compute the next object on the route within less than a millisecond. (Except for the Recalculating versions of MED and Optimistic.) The difference in the efficiency of the algorithms is in the preprocessing time they require. When users initiate a route search, they may want the first object to be provided instantly, and thus, the efficiency of the preprocessing is important in many cases.

Table 1 presents the pre-processing times of the different algorithms. It shows that the greedy algorithms are the most efficient. MED is the least efficient because it requires a relatively long preprocessing step. It can be seen that the preprocessing requires significantly less time for a complete order than for no order. In general, the efficiency of the preprocessing is inversely proportional to the number of possible orders that comply with the order constraints.

6. CONCLUSION

We studied the problem of interactive route search in the presence of order constraints, for two cases. In one case, the constraints define a complete order over the types of entities that should be visited, and in the other they define a partial order. For each case, we presented three algorithms, having in mind two goals: computing an effective route (i.e., a route that is as short as possible) and doing it efficiently (i.e., finding the next object on the route as quickly as possible). The Greedy algorithm is the most efficient, yet the route it computes is the least effective. The MED algorithm, in contrast, provides the most effective route; however, its efficiency is the lowest. The Optimistic algorithm is a compromise that provides a route with effectiveness and efficiency that are between those of MED and Greedy. The differences between the running times of the three algorithms are just in the preprocessing phase. The time needed to find the next object is about the same in all of them (less than 1 millisecond). If efficiency is important, then the best may be a hybrid approach that determines the first object using the Greedy algorithm, and then switches to the MED (or Optimistic) algorithm in order to find subsequent objects. The time it takes the user to get to the first object is more than enough for completing the preprocessing. Thus, the hybrid approach is both efficient and effective.

Future work includes studying the effect of different types of indexes on the efficiency and scalability of our algorithms.

7. REFERENCES

[1] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann, *The multi-rule partial sequenced route query*, GIS, 2008, pp. 1–10.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, third edition*, The MIT Press, 2009.

[3] X. Huang and C.S. Jensen, *In-route skyline querying for location-based services*, W2GIS, 2004, pp. 120–135.

[4] Y. Kanza, R. Levin, E. Safra, and Y. Sagiv, *An interactive approach to route search*, GIS, 2009.

[5] Y. Kanza, E. Safra, and Y. Sagiv, *Route search over probabilistic geospatial data*, SSTD, 2009, pp. 153–170.

[6] Y. Kanza, E. Safra, Y. Sagiv, and Y. Doytsher, *Heuristic algorithms for route-search queries over geographical data*, GIS, 2008, pp. 1–10.

[7] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.H. Teng, *On trip planning queries in spatial databases*, SSTD, 2005, pp. 273–290.

[8] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi, *Optimal sequenced route query*, VLDBJ **17** (2008), no. 8, 765–787.

[9] M. Terrovitis, S. Bakiras, D. Papadias, and K. Mouratidis, *Constrained shortest path computation*, SSTD, 2005, pp. 181–199.

APPENDIX

A. NAIVE GREEDY HEURISTIC

The *naive greedy heuristic* works as follows. In each iteration, this heuristic chooses the candidate object that is closest to the current location l. Note that l is s in the first iteration, and l is some o_k in subsequent iterations. Formally, when $q\text{-}unsat_R(\sigma) = \emptyset$, all the queries of $\mathcal{Q}$ have been satisfied, and hence, t is the next location and the computation ends. When $q\text{-}unsat_R(\sigma) \neq \emptyset$, the naive greedy heuristic chooses a candidate object o' that is nearest to l, namely, $o' \in candidates_R(\sigma)$ and

$$dist(l, o') = \min \{ dist(l, o) \mid o \in candidates_R(\sigma) \}.$$

B. ORIENTED GREEDY HEURISTIC

The naive greedy heuristic is simple and efficient. However, it suffers from the drawback of ignoring the location of the target t. Consequently, it may compute a route that drifts far away from t and is unnecessarily long, due to the distance from the last object to t. A possible solution is to choose the next object o' based on the combined distance of o' from both the current location and t. This approach is likely to compute a route in the general direction toward t. But the route might progress too fast toward t, that is, within a few steps, the route will reach objects near t, even when there are many relevant objects in the vicinity of s and only a few near t.

The *oriented greedy heuristic* is aimed at solving the above problems by choosing the next object o' so that it will be near the current location as well as close to the straight line from s to t. In order to do so, the algorithm computes for each candidate object o', the sum $dist(l, o') + dist(s, o') + dist(o', t)$, where l is the current location. Then the algorithm chooses a candidate object that minimizes this sum.

C. DISTANCE TO TARGET

The algorithm *DistanceToTarget* of Figure 2 computes $dist\text{-}t(o)$ for all the objects $o \in A_i$, for $1 \leq i \leq m$. It is based on Dijkstra's shortest-path algorithm [2]. The algorithm iterates through the answer sets in reverse order, that is, from A_m to A_1. For each object o of A_m, the loop of Lines 1–2 computes $dist\text{-}t(o)$, which is simply the distance from o to t. Line 3 iterates through the remaining answer sets. The loop of Lines 4–5 computes $dist\text{-}t(o)$ for all objects o of A_i using the values computed for A_{i+1}. In particular, $dist\text{-}t(o)$ is the minimum of the sum $dist(o, o') + dist\text{-}t(o')$ over all $o' \in A_{i+1}$.

```
DistanceToTarget ($A_1, \ldots, A_m, t$)
Input: Target location $t$, answer sets $A_1, \ldots, A_m$ ordered according to the order defined by $C$
Computes: For each object $o \in A_i$, the minimal distance of a route when starting at $o$, continuing to an object of $A_{i+1}$, then to an object of $A_{i+2}$ and so on until getting to an object of $A_m$ and ending at $t$.
1: for each $o \in A_m$ do
2:     $dist\text{-}t(o) \leftarrow dist(o, t)$
3: for $i = m - 1$ downto 1 do
4:     for each $o \in A_i$ do
5:         $dist\text{-}t(o) \leftarrow \min_{o' \in A_{i+1}} (dist(o, o') + dist\text{-}t(o'))$
```

Figure 2: Computing the distance-to-target values

D. MED FOR COMPLETE ORDER

D.1 The Algorithm

The MED algorithm for a complete order is presented in Figure 3. Line 3 uses a subroutine that returns an array E, such that for all objects o, the entry $E[o]$ is an estimation of the expected distance covered by the shortest route that satisfies all the remaining search queries, starting with the one that corresponds to o. (The computation of E is described in Section 4.4.1 and the pseudo-code is given in Figure 4.) The loop of Line 5 (of Figure 3) iterates over the answer sets A_i in the order of increasing i. The loop of Line 7 iterates until it finds an object of A_i that satisfies Q_i; if eventually none is found, then the algorithm terminates in Line 9. Line 10 chooses the next object o to be the one that minimizes the sum of the distance from the current location to o plus the expected length of a route from o to t. If o satisfies its corresponding query, then the algorithm proceeds to the next iteration of Line 5; otherwise, o is deleted from A_i and another iteration of Line 7 is done.

D.2 The Penalty

In Section 4.4.1 we defined the penalty for a failure in the last object of A_i as $penalty(A_i) = \Sigma_{j=1}^{k-1} dist(o^i_j, o^i_{j+1})$. We explain now why the penalty is defined this way. Essentially, a penalty that is either too small or too large will cause MED to be ineffective.

When the penalty is too small, the algorithm will generate a route that skips many objects, and hence, is likely to fail in satisfying all the queries. To see that, suppose that o is the last object of A_i according to $\prec$, and the penalty is equal to zero. In this case, o would be chosen early in the MED algorithm (in Line 10 of Figure 3) and the generated route will skip many objects of A_i.

When the penalty is too large, the algorithm may generate a route that does not skip objects, and thus, goes even via objects that are very far from s and t. This may cause the generated route to be too long. Intuitively, the reason for this is that in the computation of the expected distance, the penalty is multiplied by the probability of a failure in all the visited entities. As the penalty gets larger, even a small drop in the probability of a failure causes a relatively large decrease in the expected distance. Therefore, it is more likely that the algorithm will compute a route that does not skip objects (since the more objects are visited, the lower is the probability of a failure.)

We tested various definition of the penalty, and the one we have chosen proved to be the best.

E. PHANTOM OBJECTS

The optimistic approach computes the exact minimal distance (using the algorithm *DistanceToTarget* of Figure 2) in the case of a complete order. As noted earlier, in the case of partial orders, the optimistic approach computes only an estimation of the minimal distance. The reason for that is that it takes into account the search queries that have already been satisfied, but not the possibility that some of

```
MED ((s, t, 𝒬, C), D, ≺)
Input: Start location $s$, target location $t$, search queries $Q_1, \ldots, Q_m$ ordered according to $C$, a dataset $D$, an order $\prec$ over $D$
Output: The next object to be visited
1:  if $\mathcal{Q}$ is empty then
2:      return $t$
3:  call ComputeExpLen $(o, E, (s, t, \mathcal{Q}, C), D, \prec)$
4:  $curr \leftarrow s$
5:  for $i = 1$ to $m$ do
6:      $found \leftarrow false$
7:      while not $found$ do
8:          if $A_i = \emptyset$ then
9:              return "the route cannot be completed"
10:         $o \leftarrow \operatorname{argmin}_{o \in A_i}(dist(curr, o) + E[o])$
11:         provide $o$ to the user and get a feedback
12:         $curr \leftarrow o$
13:         if $o$ does not satisfy $Q_i$ then
14:             remove $o$ from $A_i$
15:         else
16:             $found \leftarrow true$
```

Figure 3: MED for route-search queries in which C defines a complete order

the visited objects have failed. The values of the minimal distances are computed in a preprocessing step. So, when they are actually used during the construction of a path, it could be that a specific value is based on using an object that has already been visited and failed; hence, this value is only an estimation. We say that a *phantom object* is used if the choice of the next object is based on a value of the minimal distance that incorporates an object that has already been visited. The phenomenon of phantom objects can also occur in the MED algorithm for partial orders.

A simple solution to the effect of phantom objects is to do the following in each step of computing the next object to be visited. If the most-recently visited object has failed, then discard it and recalculate the estimations before determining the next object. We refer to the versions of Optimistic and MED that perform recalculation of the estimations as *Recalculating Optimistic* and *Recalculating MED*, respectively.

This solution is detrimental to the efficiency of the algorithms. Fortunately, our tests show that phantom objects are rare and that recalculating the estimations decreases the length of the produced route only by a very small amount.

F. COMPLEXITY ANALYSIS

We now analyze the complexity of the different algorithms. For interactive algorithms, the time complexity of computing an entire route is not useful because the algorithms are delayed by the need to wait for feedbacks from the user. So instead, we use the following two complexity measures. The *preprocessing complexity* is the time complexity of the computation that is required for providing the first object of the route. The *step complexity* is the time complexity of computing the next object on the route after at least one object has been computed. We analyze our algorithms according to these two measures. In our analysis, we assume that there are n objects in D and these objects are partitioned into m answer sets.

```
ComputeExpLen (E, (s, t, 𝒬, C), D, ≺)
Input: Route-search query $(s, t, \mathcal{Q}, C)$, a dataset $D$, an order $\prec$ over the objects of $D$
Output: Array $E$ such that for all $o \in D$, the entry $E[o]$ is an estimation of the expected distance from $o$ to $t$
1:  let $o^m_{k_m} \succ \cdots \succ o^m_1$ be the objects of $A_m$
2:  $E[o^m_{k_m}] \leftarrow prob(o^m_{k_m}) \cdot dist(o^m_{k_m}, t) + (1 - prob(o^m_{k_m})) \cdot penalty(A_m)$
3:  for $j = k_m - 1$ downto 1 do
4:      $o \leftarrow \operatorname{argmin}_{\{o \mid o \in A_m \wedge o^m_j \prec o\}} (dist(o^m_j, o) + E[o])$
5:      $E[o^m_j] \leftarrow prob(o^m_j) \cdot (dist(o^m_j, t)) + (1 - prob(o^m_j)) \cdot (dist(o^m_j, o) + E[o])$
6:  for $i = m - 1$ downto 1 do
7:      let $o^i_{k_i} \succ \cdots \succ o^i_1$ be the objects of $A_i$
8:      $o \leftarrow \operatorname{argmin}_{o \in A_{i+1}}(dist(o^i_{k_i}, o) + E[o])$
9:      $E[o^i_{k_i}] \leftarrow prob(o^i_{k_i}) \cdot (dist(o^i_{k_i}, o) + E[o]) + (1 - prob(o^i_{k_i})) \cdot penalty(A_i)$
10:     for $j = k_i - 1$ downto 1 do
11:         $o_{A_i} \leftarrow \operatorname{argmin}_{\{o_{A_i} \mid o_{A_i} \in A_i \wedge o^i_j \prec o_{A_i}\}} (dist(o^i_j, o_{A_i}) + E[o_{A_i}])$
12:         $o_{A_{i+1}} \leftarrow \operatorname{argmin}_{o_{A_{i+1}} \in A_{i+1}} (dist(o^i_j, o_{A_{i+1}}) + E[o_{A_{i+1}}])$
13:         $E[o^i_j] \leftarrow prob(o^i_j) \cdot (dist(o^i_j, o_{A_{i+1}}) + E[o_{A_{i+1}}]) + (1 - prob(o^i_j)) \cdot (dist(o^i_j, o_{A_i}) + E[o_{A_i}])$
```

Figure 4: Computing the expected distances

The Naive Greedy and the Oriented Greedy algorithms require no preprocessing. The computation of the first object on the route has the same time complexity as the computation of any other object on the route. In each step, all the objects of the dataset D are examined. Thus, these algorithms can be easily implemented to have $O(n)$ preprocessing complexity and $O(n)$ step complexity.

The Optimistic algorithm for the case of a complete order starts by executing a preprocessing step of computing the distance-to-target values. For each object of D, a value is computed by examining the distance from it to all the objects of the next set. Thus, the preprocessing has $O(\frac{n^2}{m})$ time complexity. The computation of an object is done by choosing an object from a set of at most n objects. Hence, the step complexity is $O(n)$.

In the case of a partial order, there can be $m!$ possible orders, and hence the preprocessing has time complexity $O(\frac{n^2}{m}m!)$. The step complexity requires checking n objects and considering at most 2^m entries in the EDT of each object. Thus, the step complexity is $O(n2^m)$.

The algorithm MED for the case of a complete order has a preprocessing step of computing the expected-distance values for the objects. First, the objects of D are sorted. The sort has $O(n \log n)$ time complexity. An expected distance is computed for each object and this is done by considering about $\frac{n}{m}$ objects of some answer set. Thus, the preprocessing complexity is $O(n \log n + \frac{n^2}{m})$. The step complexity

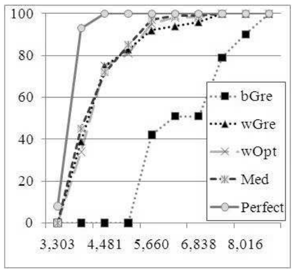

Figure 5: No order

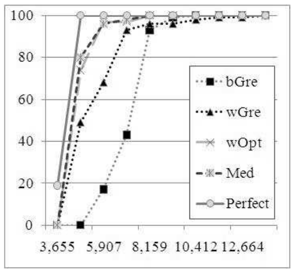

Figure 6: Partial order

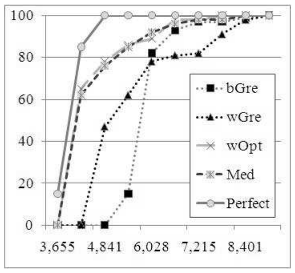

Figure 7: Complete

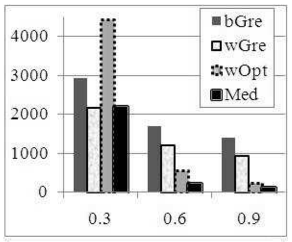

Figure 8: Comparison to the perfect result

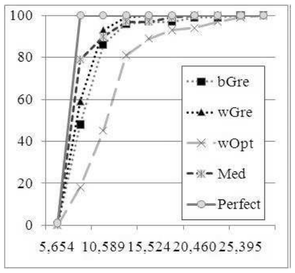

Figure 9: Mean 0.3

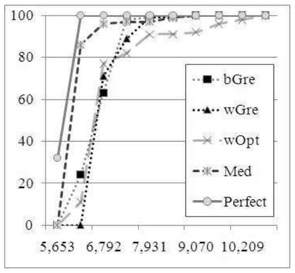

Figure 10: Mean 0.6

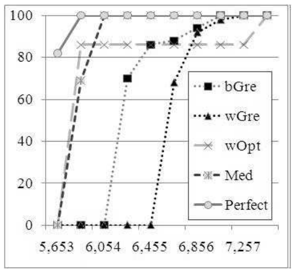

Figure 11: Mean 0.9

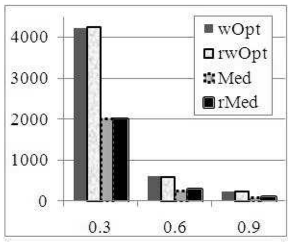

Figure 12: The effect of recalculating

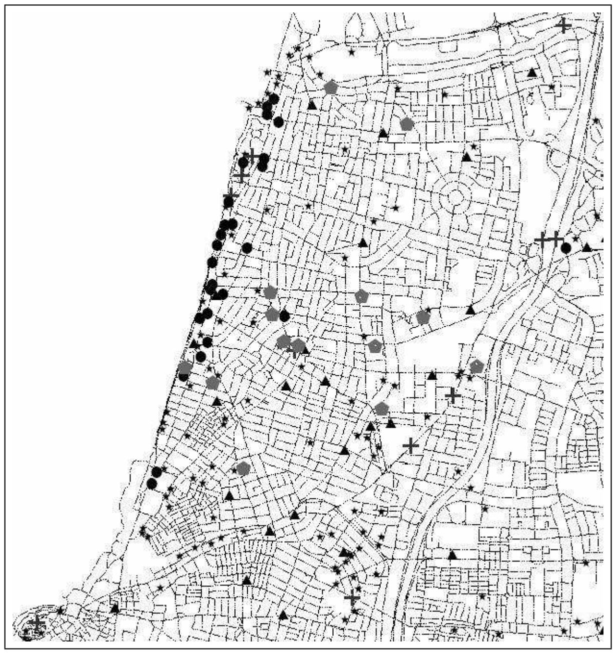

Figure 13: Map of Tel-Aviv (fragment)

requires choosing an object from an answer set, therefore, the step complexity is $O(n)$.

For constraints that define a partial order, MED needs to create EDTs and use them in each step. The preprocessing complexity requires considering $m!$ orders, and hence, it is $O(m!(n \log n + \frac{n^2}{m}))$. The step complexity is $O(n2^m)$, as for Optimistic with a partial order.

Note that in practical scenarios, the number of queries, m, is relatively small. It is reasonable to assume that in most practical cases, users will pose route-search queries of no more than ten search queries. Thus, even though the preprocessing complexity and the step complexity are exponential in m, in the case of partial orders, in practice our algorithms provide answers in an acceptable time. The experiments in the Section 5 confirm this.

G. DEFINING THE SEARCH REGION

We define the *search region* of a query as all the objects contained in a bounding ellipse whose two foci are the user location l, and the target location t. An object with location x is inside the ellipse if $|l-x|+|t-x| \leq \frac{|l-t|}{e}$, where $e \in [0,1]$ denotes the eccentricity of the ellipse, which is the ratio of the distance between the foci to the length of the major axis. The relevant objects are retrieved using an R-tree index. This method was proposed and analyzed by Li *et al.* [7] for filtering out objects that due to their location are not likely be visited by the shortest route from l to t. Thus, it reduces the number of objects being examined.

H. SETTING

The real-world data that we used in our experiments is part of a digital map, of the city Tel-Aviv, that has been generated by the Mapa company. A fragment of that map is presented in Figure 13. In our tests, we used the "Point Of Interest" (POI) layer of the map. The objects in this layer represent many different types of geographical entities. We extracted from the map 628 objects of seven different types (20 cinemas, 29 hotels, 31 pedestrian bridges, 54 post offices, 136 pharmacies, 169 parking lots and 189 synagogues). In the experiments, we tested route-search queries R where the number of search queries in $\mathcal{Q}$ is between three to seven.

In order to simulate interactive scenarios, the satisfaction of each visited object was chosen randomly, when the object was visited, according to the probability of the object. Since we wanted to prevent extreme cases, we ran every query

100 times, where in each run, different random choices were made for the objects, and the results were averaged.

Table 1: Pre-processing times, in milliseconds, for 5 search queries, over a dataset of 419 objects.

Algorithm	Full Order	Partial Order	No Order
bGre	0.6	22	115
wGre	1.6	34	167
wOpt	145	3015	16,217
Med	244	5146	26,207

I. EXAMPLES OF SPECIFIC ROUTES

We present two cases that illustrate some of the differences between our algorithms. In these two cases, we used real-world datasets, and we run our algorithms so that the results will reflect the actual behavior of the algorithms. For simplicity of presentation, there are no order constraints in the two examples of this section.

The first case compares the greedy algorithm to MED, and it shows why in many cases MED outperforms greedy. It is presented in Figure 14 .

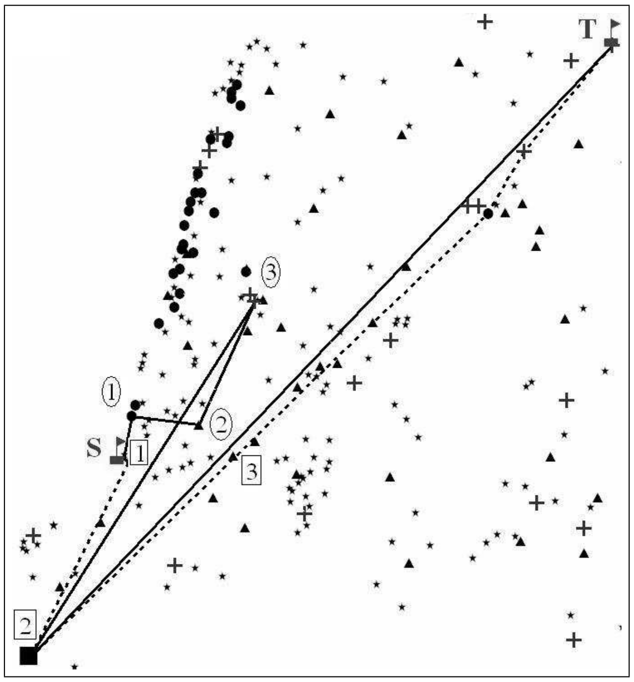

Figure 14: A scenario where the route computed by Greedy (solid line) is significantly longer than the route computed by MED (dashed line).

Example I.1. *In this example, a route search with five queries is considered. The objects to be visited are depicted by plus, star, triangle, circle and square icons. The route provided by the greedy algorithm is depicted with a solid line and some locations where the user provided a feedback are depicted as a number inside a circle. The route that MED computed is depicted with a dashed line, and the locations where the user provided a feedback are shown as a number inside a square.*

The result of one of the search queries consists of a single object, and it is depicted using a black square at the bottom left corner of the figure. Since there is only one such square, the route must go via this location.

MED "plans" the entire travel, and thus, it goes from the start location to the location of the black square. (This is also marked by the number 2 inside a square). Then, MED continues directly to the target location going via the other objects it needs to visit.

The greedy algorithm goes to objects that are near the line that connects s and t. It leads the user to the locations depicted by 1,2 and 3 in a circle. The greedy approach leads toward t till there is only one query left to satisfy—the query whose answer is the black square. This forces the route to lead back in a direction opposite to t, visit the black square and continue to t. Going back and forth due to lack of planning causes the greedy to be inefficient in such case.

The second example compares Optimistic to MED.

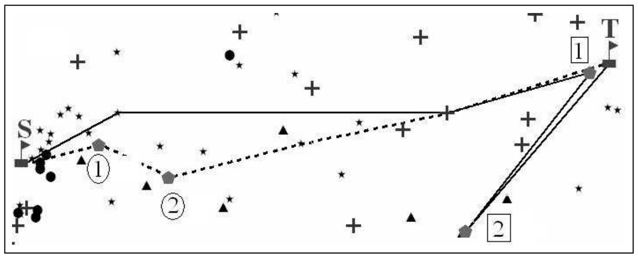

Figure 15: A scenario where the route computed by Optimistic (solid line) is significantly longer than the route computed by MED (dashed line).

Example I.2. *The scenario depicted in Figure 15 illustrates the superiority of MED over Optimistic. In this scenario, the route-search query consists of three search queries whose results are depicted by plus, star and pentagon icons, respectively.*

The pentagons represent cinemas. In this scenario, cinemas have a probability of 0.7. There is a cinema near t. Optimistic computes the shortest route and reaches that cinema (see the number 1 in square near that cinema). However, in many cases this cinema fails to satisfy the user. In these cases, the route continues to a cinema that is far from t (there is an icon of the number 2 inside a square near that cinema). So, in this scenario, Optimistic generates a route that frequently goes back and forth.

MED, on the other hand, considers the case that the cinema near t will fail and hence, it visits cinemas on the way from s to t (these cinemas are marked by 1 and 2 in a circle). If cinema 1 fails, the route continues to 2 with only a small increase in the total length, whereas for the route of Optimistic, when the first cinema fails the increase in the length of the route is large.