Engineering Shortest Regular Category-Path Queries

Le-Duc Tung, Kento Emoto, Zhenjiang Hu

A Shortest Regular Category-Path (SRCP) query is a variant of constrained shortest path queries, in which a candidate path of minimum total length has to visit a number of typed locations in a specific way according to a regular expression over the types of locations such as banks, convenience stores, police stations, etc. Types of locations are referred to as categories. This problem is important for flexible routing in road networks. In this paper, we first show that this problem can be solved by a sequence of Single Source Shortest Path (SSSP) searches; as a result, any fast SSSP algorithm can be applied to speed up its computation. Next, we progressively engineer an efficient solution that optimizes the sequence of SSSP searches. Finally, we do some experiments on the full American road network (with over 20 million vertices and nearly 60 million edges), which has shown that our solution is practical and scalable for large, real-world road networks.

1 Introduction

In road networks, locations usually belong to one or more specific categories such as bank, clothing store, shopping mall, etc. Finding shortest paths based on these categories is very useful and practical. For example, when working hour is up, on the way back home we want to buy some clothes, but we need to withdraw money in advance. Hence, we should find the shortest path from the office (o) to the home (h), visiting a Bank (B) followed by a Clothing store (C), or a Shopping mall (S). This query is reasonable because in reality there must be an ATM and a clothing store in a shopping mall. Figure 1 shows an example of a network with three categories and one of the shortest paths satisfying the above query. In real-life datasets, one category consists of up to hundreds of thousands of locations, which makes the problem difficult to solve efficiently.

Much research in recent years has focused on proposing queries to express some practical problems related to categories. Li et al. [5] introduced Trip Planning Queries to find the shortest path going from a starting location, passing through at least one location from each category in a user-specified set of categories and ending at a destination category. Rice et al. [8] proposed an exact solution for the same problem with one destination location and called it Generalized Traveling Salesman Path problem. In parallel to Li's work, Sharifzadeh et al. [11] addressed a query called Optimal Sequenced Route query in which a total order over all categories is specified. Rice et al. [9] considered a Generalized Shortest Path query that is similar to the Optimal Sequenced Route query but with one destination location. Recently, Li et al. [6] proposed another query that allows partial order constraints between different categories, e.g., a gas station must be visited before a restaurant but other locations can be visited in an arbitrary order. Although there are many queries have been proposed, there is a lack of queries that can directly express alternative routes such as the simple query shown

Le-Duc Tung, 総合研究大学院大学（総研大), Dept. of Informatics, The Graduate University for Advanced Studies (SOKENDAI).

Kento Emoto, 九州工業大学, Dept. of Artificial Intelligence, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology.

Zhenjiang Hu, 国立情報学研究所, Programming Research Laboratory, Information Systems Architecture Research Division, National Institute of Informatics (NII).

in the first paragraph.

In this paper, we discuss a novel query called the Shortest Regular Category-Path (SRCP) Query. It is an extension of the Generalized Shortest Path Query [9] in which alternation operators "|" are added to express alternative routes in the query. This extension makes the query more powerful and flexible. The example in the first paragraph can be described in the SRCP query as a triple "$\langle \mathtt{o}, \mathtt{h}, \mathtt{BC} \mid \mathtt{S} \rangle$". One naive solution is enumerating all possible total order constraints in the query and evaluating them independently. However, this naive solution produces a significant amount of repeated computations, and is not practical for large datasets. Our approach is several orders of magnitude faster. In case of Trip Planning Queries, our algorithm can produce an exact answer and its time complexity is exponential in the number of categories.

Barrett et al. [1] introduced a formal-language-constrained shortest path query, in which shortest paths are constrained by a formal language describing requirements on labels associated with the nodes/edges of the graph. The SRCP query can be seen as a special class of this query, in which constraints are regular expressions over categories of nodes. Barret's algorithm computes a product graph of the original graph and a nondeterministic finite automaton constructed from the regular expression. One advantage of this approach is that we can easily reduce the problem to a point-to-point shortest path search on the product graph. However, it becomes nontrivial to apply speedup techniques (e.g Contraction Hierarchies [3]) to the point-to-point shortest path search because the product graph is highly depended on user-defined queries. Different to their approach, we reduce the problem to a series of single source shortest path searches on the input graph and the underlying implementation of a single source shortest path search is independent of our algorithm. Consequently, we can utilize any fast algorithm for the single source shortest path search.

Our contributions are summarized as follows.

- We first introduce the SRCP query that covers a class of existing queries such as trip planning queries, optimal sequenced route queries, optimal route queries with arbitrary order constraints, generalized traveling salesman path queries, and generalized shortest path queries.
- We then establish a dynamic programming formulation for the SRCP query algorithm, which shows that the SRCP query can be answered by a series of single source shortest path searches. This is very useful because we can utilize any fast single source shortest path search (sequential or parallel search) to gain the best performance.
- Finally we progressively engineer an efficient algorithm for answering the SRCP query by using two forward and backward optimizations. Experimental results showed that the algorithm is practical to real-life datasets.

The rest of this paper is organized as follows. Section 2 gives a formal definition of the Shortest Regular Category-Path query. In Sect. 3 we introduce a basic solution based on dynamic programming and then engineer it to get an efficient algorithm for answering the SRCP query. The expressiveness of the SRCP query is also discussed in this section. A set of experiments with large datasets is showed in Sect. 4. Section 5 discusses some related works and Sect. 6 concludes the paper.

2 SRCP Queries

In this section, we formally give a definition for the SRCP query and SRCP problem. At the end of the section, we briefly discuss a naive solution that enumerates all permutations and evaluate them seperately, which is is inefficient in answering the SRCP query.

2.1 Premininaries

Definition 2.1. *(Graph and Shortest Path) Let $G = (V, E, w)$ be a weighted digraph, where V is the set of nodes in G, $E \subseteq V \times V$ is the set of edges in G, and $w : E \to \mathbb{R}_+$ is a function mapping each edge in G to a*

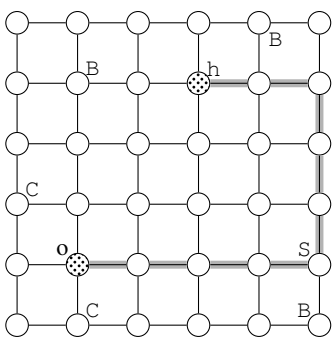


Figure 1 An example of a graph whose nodes belong to one of the categories: Bank (B), Clothing store (C), Shopping mall (S). h denotes the home, o denotes the office. The path shaded in grey is the shortest path satisfying the SRCP query "$\langle$o, h, BC | S$\rangle$"

positive, real-valued weight.

Let $P_{s,t} = \langle v_1, v_2, \ldots, v_q \rangle$ be any path in G from node $s = v_1 \in V$ to node $t = v_q \in V$, such that, for $1 \leq i < q, (v_i, v_{i+1}) \in E$. Let $cost(P_{s,t}) = \sum_{1 \leq i < q} w(v_i, v_{i+1})$ be the cost of $P_{s,t}$. A path $P'_{s,t}$ is called a shortest path *from s to t if $\forall P_{s,t} \in G$, we have $cost(P'_{s,t}) \leq cost(P_{s,t})$. The* shortest path cost*, $cost(P'_{s,t})$, is denoted by $d(s,t)$.*

Definition 2.2. *(Category) A category is a set of vertices in a graph.*

Definition 2.3. *(Simplified Regular Expression (SRE)) The syntax for SREs is:*

$$R ::= \_*C\_* \mid RR \mid R \text{ "|" } R$$

Here C is a category, i.e., a terminal symbol. $R_1 \mid R_2$ denotes the alternation, matching either R_1 or R_2. The expression $R_1 R_2$ denotes the concatenation. "_" is the combination of the wildcard (matching any terminal symbol) and the Kleene star, matching any sequence of possible categories. The concatenation binds more strongly than the alternation. For simplicity, we use "C" as an abbreviation of "_*C_*".*

2. 2 SRCP Queries

Definition 2.4. *(Shortest Regular Category-Path (SRCP) Query) Given a weighted digraph $G = (V, E, w)$, let $\{C_i \subseteq V \mid 1 \leq i \leq k\}$ be a set of categories of nodes in G, and R be an SRE over C_is. An SRCP query is represented as a triple $\langle s, t, R \rangle$, where $s, t \in V$. s is called a starting node and t a target node.*

Definition 2.5. *(Satisfying path) A path $P_{s,t} = \langle v_1, v_2, \ldots, v_k \rangle$ from s to t is said to satisfy an SRE R over a set of distinguished categories if the concatenation of categories of these nodes spells out R. Such a path is denoted by $P_{s,R,t}$.*

Definition 2.6. *(Shortest Regular Category-Path (SRCP)) Given an SRCP query $\langle s, t, R \rangle$, an SRCP is a path $P^{min}_{s,R,t}$ satisfying R, and for every path $P_{s,R,t}$ in G satisfying R, we have:*

$$cost(P^{min}_{s,R,t}) \leq cost(P_{s,R,t}).$$

We refer $cost(P^{min}_{s,R,t})$ as $d^R(s,t)$.

Definition 2.7. *(SRCP problem) An SRCP problem is to find an SRCP for a given SRCP query.*

Figure 2 shows an example SRCP query and paths satisfying the query. The input graph has five categories O, G, B, V, and Y. Nodes in the same category have the same shape and color. Because of alternations in the query, it is possible to have many optimal paths $P^{min}_{s,R,t}$ that have the same optimal cost $d^R(s,t)$. Also note that there is no requirement that two nodes in two different categories must be directly connected. For example, the optimal path $\langle s, o_1, g_1, y_1, \ldots, v_1, \ldots, t \rangle$ spells out the constraint "OGYV", however the node y_1 in Y connects to the node v_1 in V through two other nodes although Y and V are contiguous in the constraint.

A naive approach for the SRCP problem is to generate all total order constraints R_i following the input SRE R, then evaluate sub queries $\langle s, t, R_i \rangle$ independently by efficient algorithms for total order constraints (e.g. algorithms for optimal sequenced route [11] or generalized shortest path [9]), and finally take the minimum cost from results of each sub queries. This approach is easy but contains many redundant computations between the evaluations of sub queries. In the next section we show another approach to solve the problem and engineer an efficient algorithm which is several orders of magnitute

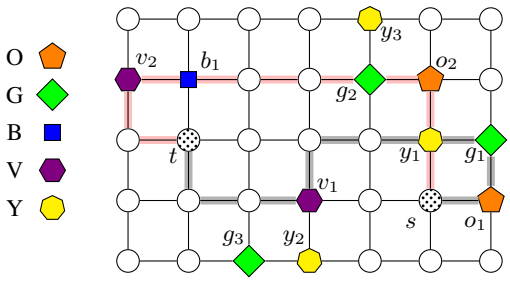


Figure 2 An SRCP query example $\langle s, t, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$. All edges have the same weight of 1. Two of the optimal solutions are shaded in grey and pink. $d^R(s,t) = 9$

faster than the naive one.

3 Engineering an efficient SRCP algorithm

In this section, we first propose a dynamic programming solution to solve the SRCP problem. The solution reduces the SRCP problem to a series of single source shortest path searches. Next, we engineer an efficient algorithm by applying some optimizations on a graph representation of the SRCP query. Finally the expressiveness of the SRCP query is discussed.

3.1 Dynamic Programming Formulation

3.1.1 SRE as Directed Acyclic Graph

First, we introduce a *constraint graph* G_R that is a directed acyclic graph (DAG) representing the semantics of SREs in SRCP queries. To avoid confusion with nodes in the input graph G (road network), we use the term "vertex" to refer a node in G_R. Vertices of G_R are object identifiers (OIDs) that are integers uniquely identifying a vertex. A Skolem function $\odot$ is used to create a fresh OID by taking two input OIDs [10]. Edges of G_R are labeled by categories. The graph G_R is constructed by a recursive function on the structures of SREs.

Given an SRE R, the following function gen_dag generates a constraint graph G_R for R. Note that, the generated graph is special in the sense that it has a source and a sink.

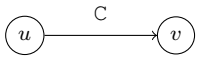


Figure 3 Singleton graph G_C

$$gen\_dag\ R = \textbf{let}\ \langle sc, G_R, sk\rangle = rec\ R\ \textbf{in}\ dag$$

where

$$\begin{aligned}
rec\ C &= \textbf{let}\ d = newEdge(C)\\
&\quad\ \textbf{in}\ \langle source(d), d, target(d)\rangle\\
rec\ (R_1R_2) &= (rec\ R_1) \ominus (rec\ R_2)\\
rec\ (R_1|R_2) &= (rec\ R_1) \oplus (rec\ R_2)\\
\langle sc_1, G_{R_1}, sk_1\rangle &\ominus \langle sc_2, G_{R_2}, sk_2\rangle\\
&= \langle sc_1, seq(G_{R_1}, G_{R_2}), sk_2\rangle\\
\langle sc_1, G_{R_1}, sk_1\rangle &\oplus \langle sc_2, G_{R_2}, sk_2\rangle\\
&= \langle sc_1 \odot sc_2, merge(G_{R_1}, G_{R_2}), sk_1 \odot sk_2\rangle
\end{aligned}$$

Given an SRE R, the recursive function rec computes a triple $\langle sc, G_R, sk\rangle$ where sc and sk are the source and sink vertex in the constraint graph G_R, respectively. For the terminal case $\mathtt{C}$, we create a singleton graph G_C containing only one edge d labeled by the category $\mathtt{C}$. The source vertex of G_C is the source vertex of d, and the sink vertex is the target vertex of d. Figure 3 shows an illustration of G_C. The function seq is to construct a new constraint graph by first replacing sk_1 in G_{R_1} and sc_2 in G_{R_2} by a new OID $w = sk_1 \odot sc_2$, then unioning these two constraint graphs. The function $merge$ is to construct a new constraint graph by first replacing sc_1 in G_{R_1} and sc_2 in G_{R_2} by a new oid $sc_{12} = sc_1 \odot sc_2$, and then replacing sk_1 in G_{R_2} and sk_2 in G_{R_2} by a new OID $sk_{12} = sk_1 \odot sk_2$, finally unioning these two constraint graphs.

To present the semantics of the whole query $Q = \langle s, t, R\rangle$, we introduce a *query graph* that is created from the constraint graph G_R by attaching an in-coming edge labeled by $\{s\}$ to the source vertex of G_R, and an out-going edge labeled by $\{t\}$ to the sink vertex of G_R. Figure 4 shows a query graph of the query $\langle \mathtt{s}, \mathtt{t}, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$. Two sets $\{s\}$ and $\{t\}$ are called dummy categories.

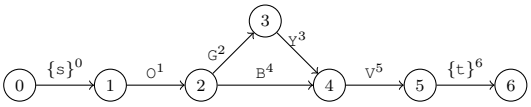


Figure 4 The query graph of the query $\langle s, t, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$. Superscripts of categories are equivalent row indices in the DP table. Integers in vertices are OIDs. The subgraph from the vertex 1 to the vertex 5 is the constraint graph corresponding to the SRE $\mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}$

3. 1. 2 Dynamic Programming Formulation

Next, we formalize a dynamic programming formulation for the SRCP problem by using the query graph. Given an SRCP query $Q = \langle s, t, R\rangle$, we establish a table X in order to store values during computation. Each row in the table corresponds to a category of an edge in the graph G_Q. Therefore, the table X has $|E(G_Q)|$ rows, and g columns where g is the maximum number of nodes of categories in the query. "$X[i, j]$" is the value for the j-th node in the category at the row i. For simplicity, we add superscripts to categories in the query to denote their row indices in the table. For example, with the user-defined query $\langle \mathtt{s}, \mathtt{t}, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$, we have $\langle \mathtt{s}^0, \mathtt{t}^6, \mathtt{O}^1\,(\mathtt{G}^2\mathtt{Y}^3 \mid \mathtt{B}^4)\,\mathtt{V}^5\rangle$.

The table X is computed according to a topological sort of the query graph G_Q as follows. For each vertex u in the topological sort, we generate a *computation step*, $in_u \rightarrow out_u$, where

$$in_u = \{ri | (v, u) \in E(Q_G), C^{ri} = l(v, u)\}$$
$$out_u = \{ro | (u, w) \in E(Q_G), C^{ro} = l(u, w)\},$$

computing values of the rows in the list out_u by using values in the rows in the list in_u, in which $l(u, v)$ is a function to get a label of the edge (u, v). Computation steps form a dynamic programing formulation for the SRCP problem. Following is the computing formulation of the computation step $in_u \rightarrow out_u$. $\forall i \in out_u$, if $i = 0$ then $X[i, j] = 0$. If $i > 0$, $X[i, j] = \min_{r \in in_u} \{\min_{0 \le l < |C^r|} \{X[r, l] + d(c_{r,l}, c_{i,j})\}\}$, where, C^r is the category corresponding to the r-th row in the DP table, $c_{i,j}$ is the j-th node in the category C^i.

Lemma 3.1. *Value $X[i, j]$ in the DP table represents the optimal cost of the SRCP of the query $\langle s, \mathtt{c}_{i,j}, R^i\rangle$ where R^i is the SRE corresponding to a subgraph of G_R that includes all paths from the source vertex of G_R to the source vertex of the edge labeled $\mathtt{C}^i$.*

For example, consider the query $\langle \mathtt{s}^0, \mathtt{t}^6, \mathtt{O}^1\,(\mathtt{G}^2\mathtt{Y}^3 \mid \mathtt{B}^4)\,\mathtt{V}^5\rangle$, the equivalent DAG is shown in Fig. 4. The value $X[3, 1]$ will represent the optimal cost of the SRCP of the query $\langle s, y_1, R^3\rangle$ in which $R^3 = \mathtt{O}^1\mathtt{G}^2$ corresponding to the subgraph having edges from the vertex 1 (the source vertex of G_R) to the vertex 3 (the source vertex of the edge $\mathtt{Y}^3$).

Proof. We prove this by induction on the sequence of computation steps $1 \le k \le N$, in which N is the number of computation steps (the number of vertices in G_R).

For the base case, where $k = 1$, then this claim is trivially true, because $X[0, 1]$ is the optimal cost for the query $\langle s, \mathtt{c}_{0,1}, R^0\rangle = \langle s, s, \{s\}\rangle$ which has $d^R(s, s) = 0$.

For the induction step, $k > 1$. Let u be the vertex in G_R that generates the k-th computation step. For a set of vertices V_u that are before the vertex u in the topological sort of G_R, let E_u be a set of edges related to vertices in V_u, and ps be a set of row indices of categories on the edges in E_u. Our induction hypothesis assumes that this claim holds true for all values $X[i, \bullet], i \in ps$. Let consider the $(k + 1)$-th computation step generated by the vertex v in G_R, that is $in_v \rightarrow out_v$, in which in_v and out_v is the set of row indices of all in-coming and out-going edges of the vertex v, respectively. It is clear that $in_v \subseteq ps$. Since each $c_{i,j}$ is a member of category C^i, $i \in out_v, j = 0 \ldots |C^i|$, it suffices to find the shortest path cost from nodes in categories C^r, $r \in in_v$ to $c_{i,j}$. It follows from our induction hypothesis that the value of $X[i, j]$ is computed by $\min_{r \in in_v} \{\min_{0 \le l < |C^r|} \{X[r, l] + d(c_{r,l}, c_{i,j})\}\}$. □

Corollary 3.2. *Let m be the row index of the dummy category of $\mathtt{t}$. Value $X[m, 0]$ represents the cost of SRCP of the query $\langle s, \mathtt{t}, R\rangle$.*

Figure 5 shows a table contained costs during the com-

putation of the query $\langle \mathtt{s}^0, \mathtt{t}^6, \mathtt{O}^1\,(\mathtt{G}^2\mathtt{Y}^3 \mid \mathtt{B}^4)\,\mathtt{V}^5\rangle$. The order of computation steps is as follows.

1st step:	$[0]$	$\rightarrow [1]$
2nd step:	$[1]$	$\rightarrow [2,4]$
3rd step:	$[2]$	$\rightarrow [3]$
4th step:	$[3,4]$	$\rightarrow [5]$
5th step:	$[5]$	$\rightarrow [6]$

The optimal cost $d^R(s,t) = 9$ of the query is stored at the last row (index = 6) of the table.

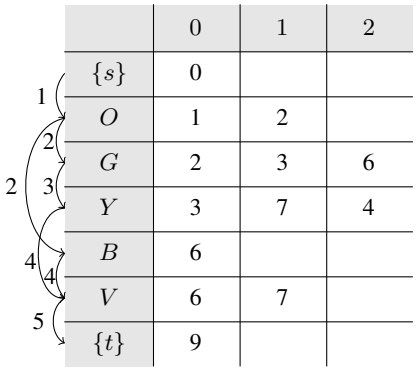
3.1.3 Single Source Shortest Path (SSSP) search

A computation step "$xs \rightarrow ys$" is to compute values in rows in the list ys by using rows in xs. Let U_{xs} be a union of categories corresponding to rows in xs and U_{ys} be a union of categories corresponding to rows in ys, then the step "$xs \rightarrow ys$" is equivalent to computing values for the nodes in category U_{ys} from values of the nodes in category U_{xs}. This can be done by using a many-to-many shortest path search from the nodes in U_{xs} to the nodes in U_{ys}. However, such a many-to-many search leads to many redundant computations due to repeatedly visiting the input graph. By creating a super-node s' and edges from s' to the nodes u in U_{xs} with weights being values of u in the table [9], we can efficiently compute values for the nodes in U_{ys} by using a single shortest path search from s' until all nodes in U_{ys} are settled (Assume that we use Dijkstra algorithm).

Theorem 3.3. *Given a weighted digraph $G = (V, E, w)$ and an SRCP query $Q = \langle s, t, R\rangle$. Let $G_R = (V_R, E_R)$ be a constraint graph of R and T_{sssp} be the complexity of a single source shortest path search, the cost of our algorithm is $O(|V_R|T_{sssp})$, and the space complexity of the algorithm is $O(|E_R|)$.*

Corollary 3.4. *Given a weighted digraph $G = (V, E, w)$ and an SRCP query $Q = \langle s, t, R\rangle$. Let $G_R = (V_R, E_R)$ be a constraint graph of R. When a Dijkstra algorithm with a Fibonacci heap is used for SSSP searches, our algorithm answers Q in the time complexity of $O(|V_R|(|E| + |V|log|V|))$.*

One advantage of this approach is that it is independent of the underlying SSSP search. We can use any fast SSSP algorithm to implement, such as Contraction Hierarchies technique [3][9], Delta-Stepping [7], PHASE [2], etc. This is useful because we can apply different efficient fast SSSP algorithms for different kinds of graphs (road networks, social networks, etc.)



	0	1	2
$\{s\}$	0		
O	1	2	
G	2	3	6
Y	3	7	4
B	6		
V	6	7	
$\{t\}$	9		

Figure 5 A table of shortest path costs for the query in Fig. 2. The first column contains names of categories. The first row contains indices of nodes in a category. Each row contains shortest path costs from s to a node in a category via some previous categories. Curve arrows on the left indicate computation steps and its orders.

Optimization is trying to reduce the number of vertices N in the constraint graph G_R, because this will reduce the number of computation steps in the algorithm. First, N depends on the user-defined query. For example, two queries $\langle \mathtt{s}, \mathtt{t}, (\mathtt{OGYV} \mid \mathtt{OBV})\rangle$ and $\langle \mathtt{s}, \mathtt{t}, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$ have the same meaning, but the former needs 6 computation steps and the latter needs only 5. Second, consider the Trip Planning Query that is to find the shortest path going through at least one point in each category of a given set of categories $\mathtt{C} = \{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_k\}$. It can be presented in SRCP query as $\langle \mathtt{s}, \mathtt{t}, R_1 \mid R_2 \mid \ldots \mid R_{k!}\rangle$ where R_is are permutations of the set $\mathtt{C}$, $i = 1 \ldots k!$. In this case, N will be $(k+1)k!$ which is not practical even for small k (e.g. $k = 5$. See Sect. 4 for more discussion). The next section will discuss how to engineer an efficient algorithm for answering the SRCP query.

3.2 Optimizing the Query Structure

Optimization is to reduce the size of the query graph G_Q. In particular, there are two measures for the size of the query graph: the number of vertices and the number of edges. The number of vertices affects the time complexity and the number of edges affects the space complexity which is the size of the dynamic programming table. Here, we focus on the problem of optimizing the time complexity, therefore the problem can be defined as finding a graph with the minimum number of vertices that generates exactly the same total order constraints as a given query graph.

The query graph is a subclass of non-deterministic finite state automatons (NFA) [4]. It is well known that NFA minimization is computationally hard, and cannot in general be solved in polynomial time. There exists a well-known algorithm for minimizing NFAs [4]. The algorithm computes a minimal equivalent deterministic finite automaton (DFA) with respect to the number of vertices. It consists of two steps: determinization and minimization. The determinization step is to compute a DFA from an NFA, then the DFA is minimized to get a minimal DFA.

Ström [13] has proposed two simplified algorithms for the two steps determinization and minimization in the case of word graphs which represent a set of hypotheses in speed recognition system. A word graph is a DAG with exactly one source vertex and one sink vertex. In this section we applied these algorithms to optimize the query graph which has a similar structure to word graphs.

3.2.1 Forward optimization (determinization)

A deterministic finite state automaton (DFA) has a property that, given a sequence of categories, there is at most one path in the DFA that generates it. The key to the determinization algorithm is to identify the correspondences between a set of vertices in the original graph and a vertex in its DFA graph. This leads to an exponential computation because there are 2^N sets of vertices in a graph of N vertices. However, the algorithm can be simplified in the case of the query graph that is a DAG with a single source vertex and a single sink vertex. It starts from the source vertex of G_Q, considers vertices coming from a common vertex and the same edges, groups them to create a new vertex in a deterministic query graph DG_Q, and ends as reaching the sink vertex. We call this process is a *forward optimization*.

Algorithm 1 describes the forward optimization. The idea is scanning the query graph G_Q from its source vertex to its sink vertex, and grouping vertices that come from the same vertex with the same categories. A vertex in the deterministic query graph DG_Q is correspondent to a set of vertices in G_Q. Initially, the graph DG_Q contains only the source vertex of G_Q. For each vertex n_x in the DG_Q and each categories c, we find a set of vertices n_y in G_Q that can be reached from a vertex in n_x (see the line 6 in Alg. 1). If n_y is not existing in DG_Q, then we have to add it to DG_Q before adding an edge from n_x to n_y with the label c to DG_Q. The vertex n_y will be added to a set $next$ for the next loop. The problem is how to store the sets of vertices in DG_Q in an efficient way which supports "looking" and "comparison". Here we use a hash-table, and sets are sorted before inserting to the hash-table.

Figure 9 describes an example of the forward optimization for the query $\langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1\mathtt{C}_4\mathtt{C}_6\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_4\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_5\mathtt{C}_7\ \mathtt{C}_9 \mid \mathtt{C}_2\mathtt{C}_8\mathtt{C}_9 \mid \mathtt{C}_3\mathtt{C}_8\mathtt{C}_9 \rangle$. We start from the vertex 0, at this time, the graph DG_Q contains only one vertex created from the vertex 0. Because there is only one edge from the vertex 0, therefore a new vertex in DG_Q contains only the vertex 1. In the next step, since there are 3 outgoing edges of the same category $\mathtt{C}_1$ from the vertex 1 in the graph G_Q, we can group target vertices into one set $\{2, 3, 4\}$ and create a new vertex in DG_Q. Two vertices 7 and 8 in G are also coming from vertices in the same set $\{2, 3, 4\}$ with the same categories $\mathtt{C}_4$, thus we also group them to create a new vertex in DG_Q, This procedure is performed until reaching the sink vertex of G_Q. Note that vertex ids in the graph DG_Q are different with those in the graph G_Q.

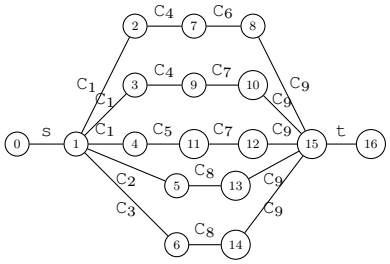


Figure 6 The query graph of the query

$\langle s, t, \mathtt{C}_1\mathtt{C}_4\mathtt{C}_6\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_4\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_5\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_2\mathtt{C}_8\mathtt{C}_9 \mid \mathtt{C}_3\mathtt{C}_8\mathtt{C}_9 \rangle$

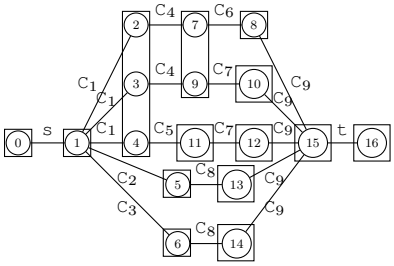


Figure 7 Vertices need to be grouped

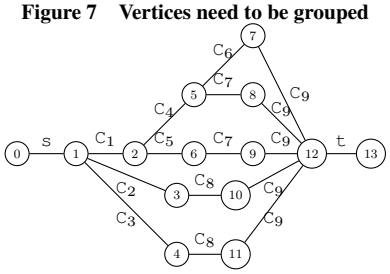


Figure 8 The deterministic query graph (vertices have been re-indexing)

Figure 9 Forward optimization.

Algorithm 1: Determinization(G_Q, DG_Q)

input : A query graph G_Q

output : A deterministic query graph DG_Q

invariant : $l(u, v)$ returns a label of the edge (u, v)

```
/* sc_G is the source vertex of G_Q */
```

1 $n_0 \leftarrow \{sc_{G_Q}\}$; $next \leftarrow n_0$;
2 **while** !$(next.empty)$ **do**
3 $\_next \leftarrow \{\}$;
4 **foreach** $n_x \in next$ **do**
5 **foreach** *category* $\mathtt{C} \in G_Q$ **do**
6 $n_y \leftarrow \{v \mid (u, v) \in E(G_Q), u \in n_x, l(u, v) == \mathtt{C}\}$;
7 **if** $n_y.empty$ **then**
8 *continue*;
9 **else**
10 **if** $n_y \notin V(DG_Q)$ **then**
11 add n_y to $V(DG_Q)$;
12 add n_y to $\_next$;
13 **end**
14 add (n_x, n_y) with label $\mathtt{C}$ to $E(DG_Q)$;
15 **end**
16 **end**
17 **end**
18 $next \leftarrow \_next$;
19 **end**

3. 2. 2 Backward optimization (minimization)

This optimization is applied to the deterministic graph that is the result from the forward optimization. If *multiple vertices* go to the same vertex with the same *set* of categories, then we can group them into one vertex. Because the deterministic query graph is a DAG in which edges go from the source vertex towards the sink vertex, we will process vertex from the sink vertex util reaching the source vertex. We call this process *backward optimization*. This process is the same as minimizing a DFA because it merges any pair of vertices that generate exactly the same sequence of categories.

Algorithm 2 shows the implementation of our backward optimization. We initialize the minimal deterministic query graph MDG_Q with the sink vertex in the input deterministic query graph DG_Q. Then we scan the graph DG_Q from the sink vertex back to the source vertex to add new equivalent vertices to the graph MDG_Q. For each pair of vertices, what we need to compare is the categories on the out-going edges and the vertices that they go to (see lines 11-20 in Alg. 2). For each vertex,

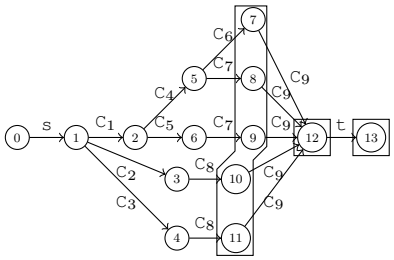


Figure 10 Step 1, 2, 3

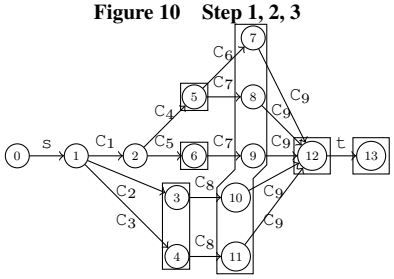


Figure 11 Step 4

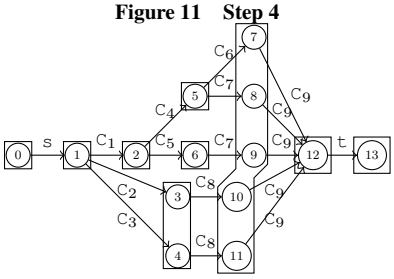


Figure 12 Step 5, 6, 7

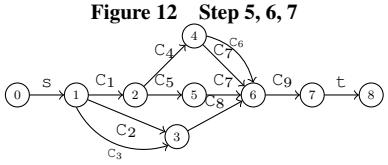


Figure 13 Minimal deterministic query graph

Figure 14 Backward optimization

we maintain a set of tuples of out-going edge's label and equivalent target vertex. If a group of vertices has the same set of categories and for each category they flow to the same vertex, then we merge them to create a new vertex in the graph MDG_Q. Figure 14 describes a process of our backward optimization.

3. 2. 3 Optimization of the DP table

The number of rows in the DP table in our solution is equal to the number of edges in the query graph G_Q. Because the number of elements in each row is the number of nodes in the equivalent category, the size of the DP table becomes large when the query contains a "long" total order constraints, leading to out-of-memory errors. Therefore we need to efficiently manage the DP table.

Algorithm 2: Minimization(DG_Q, MDG_Q)

input : A deterministic query graph DG_Q
output : A minimal deterministic query graph MDG_Q
invariant : $l(u, v)$ returns a label of the edge (u, v)

/* sk_{DG_Q} is the sink vertex of DG_Q */
$n_0 \leftarrow \{sk_{DG_Q}\}$; $next \leftarrow n_0$;
while !$(next.empty)$ **do**
  $\_next \leftarrow \{\}$;
  $n_u \leftarrow \{\}$;
  foreach $n_y \in next$ **do**
    foreach $u \in n_y$ **do**
      $\rho(u) \leftarrow \{\langle \mathtt{C}, n_y \rangle \mid (u, v) \in E(DG_Q), v \in n_y, \mathtt{C} \leftarrow l(u, v)\}$;
      add u to n_u;
    end
  end
  foreach $u \in n_u$ **do**
    $n_x \leftarrow \{v \mid v \in n_u, \rho(v) == \rho(u)\}$;
    if $n_x \notin V(MDG_Q)$ **then**
      add n_x to $V(MDG_Q)$;
      add n_x to $\_next$;
    end
    foreach $\mathtt{C} \in \{l(u, v) | (u, v) \in DG_Q, v \in n_y\}$ **do**
      add (n_x, n_y) with label C to $E(MDG_Q)$;
    end
  end
end
$next \leftarrow \_next$;

One solution is creating the DP table dynamically. As discussed earlier, the DP table is constructed according to a topological sort of the query graph of a query. When considering a node in that order, in-coming edges are used to compute values for rows corresponding to outgoing edges, and never used again. Thus, after each computation step, we need not store rows corresponding to in-coming edges.

3.3 Expressiveness

In this section, we will show how to use the SRCP query to express some existing queries of total order constraints or partial order constraints.

3.3.1 Generalized Shortest Path (GSP) Queries

This query is to find the shortest path from a starting point to a destination point, passing at least one point from each of a set of specified categories in a specified order [9]. A GSP is expressed in our SRCP query as follows.

$$\langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1\mathtt{C}_2 \ldots \mathtt{C}_k \rangle$$

where $\mathtt{s}$ and $\mathtt{t}$ are the starting point and destination point, respectively.

3.3.2 Optimal Sequenced Route (OSR) Queries

An OSR query is to find the shortest path starting from a given point and passing through a number of categories in a particular order [11]. This query is different with the GSP query in the sense that the destination is not a point but a category. To express this query in our SRCP query, we create an artificial destination node t' in the input graph, and add edges with the weight of 0 from nodes in the last category of the order constraints to t'. The SRCP query is then as follows.

$$\langle \mathtt{s}, \mathtt{t}', \mathtt{C}_1\mathtt{C}_2 \ldots \mathtt{C}_k \rangle$$

3.3.3 Trip Planning Queries/Generalized Traveling Salesman Path Problem Queries (TPQ/GTSPP)

A trip planning query [5] or generalized traveling salesman path problem query [8] is to find the shortest path from a starting point to a destination point that passes through at least one point from each of a set of categories (there is no specific order specified in the query). A TPQ/GTSPP query with a set of k categories is written in our SRCP query as follows.

$$\langle \mathtt{s}, \mathtt{t}, R_1 \mid R_2 \mid \ldots \mid R_{k!} \rangle$$

where R_is ($i = 1 \ldots k!$) are permutations of the set $\mathtt{C} = \{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_k\}$.

3.3.4 Optimal Route Queries with Arbitrary Order Constraints

This query is to find the shortest path that starts from a starting point and covers a user-specified set of categories (e.g. {gas station, museum, park, restaurant}) [6]. However, here the user can specify order constraints between some specific categories of the set, e.g. a gas station must be visited before a restaurant, while other categories can be visited in an arbitrary order. Such order constraints are expressed in a *visit order graph*, in which each vertex is a category, an edge from a category $\mathtt{C}_i$ to a category $\mathtt{C}_j$ denotes that $\mathtt{C}_i$ must be visited before $\mathtt{C}_j$.

To express the optimal route queries with arbitrary order constraints in our SRCP query, there are two things need to be done. First, we need generate total order constraints from the visit order graph, then put them together by using alternation operators in the SRCP query. A simple way to generate the total order constraints is first enumerating all permutations of the set of categories, and then filtering out permutations that do not satisfy constraints in the visit order graph. Second, we need to create an artificial destination node t' for the SRCP query, which is done by adding edges with the weight of 0 from nodes in all categories in the set to t'. The SRCP query is finally as the follows.

$$\langle \mathtt{s}, \mathtt{t}', R_1 \mid R_2 \mid \ldots \mid R_l \rangle$$

where R_is ($i = 1 \ldots l$) are total order constraints satisfying the visit order graph.

4 Implementation and Evaluation

We implemented a framework[†1] to answer the SRCP query. The overview of our framework is showed in the

†1 http://www.prg.nii.ac.jp/members/tungld/srcp-July2014.tar.gz

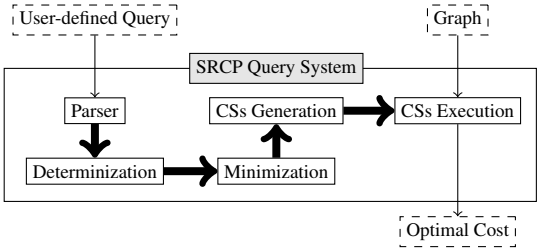


Figure 15 The overview of our framework

Fig. 15. The system takes an SRCP as input, parses it to get a graph of the query, then optimizes the graph by two steps "determinization" and "minimization". Next, the system will generate a sequence of computation steps. Each computation step is executed by a single source shortest path search. For simplicity, we just compute the optimal cost for the optimal path satisfying the query. However, one can extract the optimal path by keeping traces of computation steps.

For the implementation of a single source shortest path, we used a fast algorithm proposed by Rice [9], which has been engineering based on the speed-up technique called Contraction Hierarchies. Contraction Hierarchies (CH) technique currently is one of the fastest speed-up technique for shortest path problem. Its idea is preprocessing a graph by adding shortcuts to the graph so that shortest path costs are preserved. Shortcuts are then intensively used by a bidirectional Dijkstra algorithm to speed up the shortest path search.

4.1 Environment

All our experiments have done on a Macbook Pro machine which has a 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3 memory, clang-503.0.40 (based on LLVM 3.4svn). Programs were compiled with optimization level 3. We used a library of contraction hierarchies written by Robert Geisberger[†2] in C++ to access to contraction hierarchies of a graph.

Experiments were performed with a graph of the Full USA road network, having $23,947,347$ nodes and $58,333,344$ edges. We borrow the graph from the benchmarks of the 9th DIMACS implementation challenge [†3]. It takes about 25 minutes to preprocess the graph using CH technique.

The following programs are implemented and used in our experiments to compare results of discussed algorithms.

- **gsp**: the algorithm proposed by Rice [9] to answer the Generalized Shortest Path Query that uses total order constraints $\langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1\mathtt{C}_2 \ldots \mathtt{C}_k \rangle$.
- **perm**: a naive algorithm to answer the SRCP query by evaluating sub-queries for all total order constraints, then taking the minimal cost from the sub queries. Sub-queries are implemented by the **gsp** algorithm.
- **srcp-noopt**: our solution for the SRCP query without optimization.
- **srcp-opt**: our solution for the SRCP query with optimization.

Categories used in queries in this section will be generated randomly and they have an equal the number of nodes.

4.2 Results

4.2.1 Trip Planning Queries/Generalized Traveling Salesman Path Problem Queries (TPQ/GTSPP)

We measure the performance of our solution for the trip planning queries which are expensive queries. The Trip Planning Queries (TPQ) with k categories is written in SRCP as follows.

$$\langle \mathtt{s}, \mathtt{t}, R_1 \mid R_2 \mid \ldots \mid R_{k!} \rangle$$

where R_is are permutations of the set $\mathtt{C} = \{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_k\}, i = 1 \ldots k!$.

First, we fix the number of categories being 5, and then change the size of categories. The naive solution that generates all permutations of categories requires 720

†2 http://algo2.iti.kit.edu/source/contraction-hierarchies-20090221.tgz

†3 9th DIMACS implementation challenge: shortest paths. 2006. http://www.dis.uniroma1.it/challenge9/.

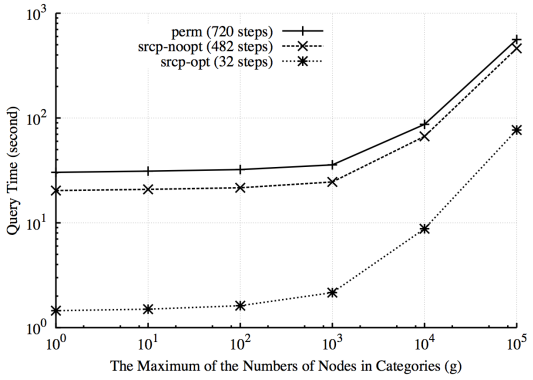


Figure 16 TPQ/GTSP queries. Vary the size of categories, fix the number of categories ($k = 5$)

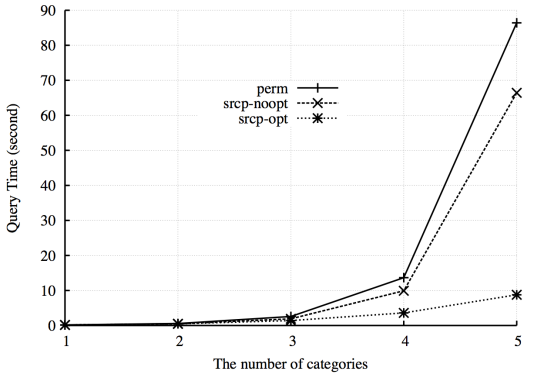


Figure 17 TPQ/GTSP queries. Vary the number of categories, fix the size of categories ($g = 10,000$)

computation steps ($(5 + 1) * 5!$). Without optimization, our algorithm generates 482 computation steps which is nearly half of that of the naive solution. By using optimizations, the number of computation steps reduces to 32 (2^5) that is more practical. Performance of algorithms is represented in the Fig. 16. It shows that our algorithm is quite scalable when the size of categories is increased.

Next, we will see the effect of the number of categories on the performance of the query. We fix the size of each category and change the number of categories in the query. As indicated by Fig. 17, we see significant improvements in query times when the number of categories is increased. Although the naive solution can reduces the number of computation steps twice, it is still following an

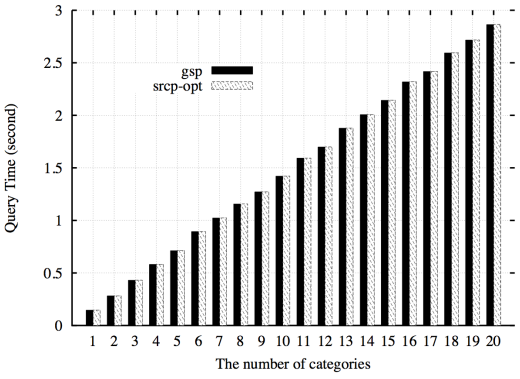


Figure 18 Optimal Sequenced Route Queries/Generalized Shortest Path Queries ($g = 10,000$)

factorial time. This experiments show the important role of optimizations in our solution.

It should also be noted that the performance of our solution is proportional to the number of computation steps. Although different computation steps have different lists of "input" categories and different lists of"output" categories, they have quite similar performance.

4.2.2 Optimal Sequenced Route Queries/Generalized Shortest Path Queries (OSR/GSP)

To see the performance of our algorithm when answering the query GSP, we compare it to the algorithm proposed by Rice [9]. As can be seen in the Fig. 18, two algorithms have the same performance when the number of categories is changed. This is easy to understand because for this query our algorithm leads to the same dynamic programing table as that of **gsp** algorithm. Moreover, there is no improvement on the structure of the query when applying optimizations, thus the overhead is very small.

4.2.3 Queries with multiple levels of options

To see the impact of alternation operators ($|$) for a given query on the performance of our optimal algorithm, we do experiments with queries which have different levels of options. We fix the number of categories in a query. Starting with a query without options, we insert one "$|$" operator to make it having one level, then insert a new one "$|$" to make one more level, and so on. In particular,

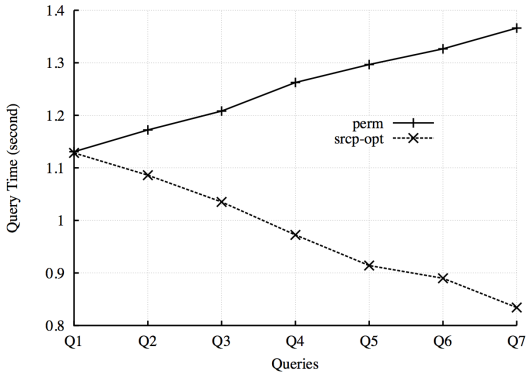


Figure 19 Queries with multiple levels of options

($g = 10,000$)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
perm	9	10	11	12	13	14	15
srcp-opt	9	8	7	6	5	4	3

Table 1 The number of computation steps in each query

we use the following queries.

$Q1 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1\mathtt{C}_2\mathtt{C}_3\mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8 \rangle$

$Q2 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid \mathtt{C}_2\mathtt{C}_3\mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8 \rangle$

$Q3 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid (\mathtt{C}_2 \mid \mathtt{C}_3\mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8) \rangle$

$Q4 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid (\mathtt{C}_2 \mid (\mathtt{C}_3 \mid \mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8)) \rangle$

$Q5 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid (\mathtt{C}_2 \mid (\mathtt{C}_3 \mid (\mathtt{C}_4 \mid \mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8))) \rangle$

$Q6 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid (\mathtt{C}_2 \mid (\mathtt{C}_3 \mid (\mathtt{C}_4 \mid (\mathtt{C}_5 \mid \mathtt{C}_6\mathtt{C}_7\mathtt{C}_8)))) \rangle$

$Q7 = \langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1 \mid (\mathtt{C}_2 \mid (\mathtt{C}_3 \mid (\mathtt{C}_4 \mid (\mathtt{C}_5 \mid (\mathtt{C}_6 \mid \mathtt{C}_7\mathtt{C}_8))))) \rangle$

Figure 19 shows the result. It is interesting that when there are more options in the SRCP query, our algorithm becomes faster. Looking the Table 1 that shows the number of computation steps for each query, we see that the reason of such good performance is due to the number of computation steps is decreased, leading to a decrease of the running time of our algorithm. Meanwhile, if we use the **perm** algorithm, then the number of computation steps will be increased because there are many redundant computation steps generated.

5 Related Works

The category-constrained shortest path problem is a variant of regular-language-constrained shortest path queries in which constraints are on a set of nodes in a graph instead of individual nodes/edges. There are many solutions proposed to answer such queries. Each solution is for a specific kind of constraints over categories.

Trip Planning Queries [5] is the query that has no ordered constraints. The existence of multiple choices per category makes the problem difficult to solve. The complexity of the TPQ is NP-Hard. Several approximation algorithms are proposed. These algorithms are based on nearest neighbor searches. A feasible path is formed by iteratively visiting the nearest neighbor of the last nodes added to the path from all nodes in the categories that have not been visited yet. The second one is the minimum distance algorithm, a novel greedy algorithm, which results a much better approximation bound. The algorithm chooses a set of nodes, one node per one category in the query. These nodes are chose so that the total distance from the start node to it and from it to the end node is the minimum among nodes in the same categories. The algorithm then create a path by following these nodes in nearest neighbor order. Rice et al. [8] proposed an exact solution for the Generalizes Traveling Salesman Path Problem Query (GTSPP) that is similar to TPQ. The algorithm is building a product graph of the original graph $G = (V, E)$ and a covering graph built on the power set of the query's categories. Finding the answer of the GTSPP query is finding the shortest path in the product graph. The time complexity is $2^k(|E| + |V|k + |V|log|V|)$, in which k is the number of categories in the query. The algorithm is then improved by incorporating the graph preprocessing technique called Contraction Hierarchies, resulting in the time complexity of $O(2^k(m' + |V|k)$, in which m' is the number of edges of the preprocessed graph.

In parallel to Li et al.'s work [5], Sharifzadeh et al [11] proposed the optimal sequenced route query (OSR) that is similar to TPQ but imposes a total order constraints over categories. In other words, OSR query is to find the shortest path between two given nodes that visits at least one node per each category in a specified order. They

proposed tow algorithms to operate on the Euclidean distance. The first one is LORD, a light threshold-based iterative algorithm. First, LORD uses a greedy search to find an threshold (upper-bound) for the cost of the optimal path. The greedy search is a successive nearest neighbor search from the starting node to the last category. Then, the LORD finds the optimal path in the reverse order, from the last category to the starting node. During the finding, it updates the threshold value and uses it to prune nodes that cannot belong to the optimal path. The second algorithm is R-LORD, an extension of the LORD, that uses R-tree to efficiently examine the threshold values. However, both algorithms are impractical to road networks where nearest neighbor searches are very expensive. Thus, another algorithm, progressive neighbor exploration (PNE), has been proposed in the paper. The idea of the PNE is incrementally create the set of candidate paths. At each step it needs two nearest neighbor searches: one is to expand the current best candidate path, the other is to refine that path by replacing the last node in the path by a new node.

Sharifzadeh et al. [12] introduced a pre-processing approach for the OSR query by using additively weighted Voronoi diagrams. This approach is efficient and practical compared with R-LORD algorithm, however, one of the disadvantages is that it is not flexible when requiring fixed sequences among categories. Rice et al. citeRice2013 proposed another approach using Contraction Hierarchies technique and dynamic programming for the Generalized Shortest Path (GSP) query that is the same as the OSR query. Its advantage is that it can be apply for any possible set of node categories in a query. Our work is inspired by the idea of applying dynamic programming from Rice et al.'s work.

Being close to our SRCP query is the optimal route queries with arbitrary order constraints proposed by Li et al.[6]. This query considers the partial order constraints over categories, which is described by a *visit order graph*. Two algorithms have been proposed namely Backward search and Forward search. The backward search algorithm computes the optimal paths in reverse manner similar to R-LORD algorithm [11]. However, instead of loading nodes belonging to the last category, the backward search retrieves the set of candidate nodes that may be part of the optimal path, which belong to *any categories* contained in the visit order graph. The forward search is similar to a greedy algorithm. It also use the backward search algorithm for backtracking process, eliminating some nodes that cannot be a part of the optimal path. Both algorithms have the time complexity of $O(N^2 \cdot 2^k)$, in which k is the number of categories in the visit order graph and N is the total number of nodes in the data set (road network or spatial database).

6 Conclusions and Future Works

In this paper, we have introduced the SRCP query for finding the optimal path constrained by categories. The query is very flexible to express several important problems such as trip planning queries, optimal sequenced route queries, and optimal route queries with arbitrary order constraints. We have proposed a dynamic programming based solution to solve the problem. We have shown that the problem can be solved by a sequence of single source shortest path searches. This result is important because we can use any fast single source shortest path search even with preprocessing to speed up the query. By exploiting the DAG representation of a query, we can easily derive an efficient algorithm for answering the query.

In the future, we will apply the latest parallel algorithm for single source shortest path searches, which is called *parallel hardware-accelerated shortest path trees* (PHASE) algorithm [2]. Another future work is to extend the query language so that it can express constraints on edges of the optimal path.

Acknowledgment

The authors wish to thank Akimasa Morihata for fruitful discussion of earlier work in this paper. We thank Kiminori Matsuzaki for suggesting the use of a directed

acyclic graph in analyzing queries. Also, thanks are to IPL and POPP members for their useful comments.

References

[1] Barrett, C., Jacob, R., and Marathe, M.: Formal-Language-Constrained Path Problems, *SIAM J. Comput.*, Vol. 30, No. 3(2000), pp. 809–837.

[2] Delling, D., Goldberg, A. V., Nowatzyk, A., and Werneck, R. F.: PHAST: Hardware-Accelerated Shortest Path Trees, *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, 2011, pp. 921–931.

[3] Geisberger, R., Sanders, P., Schultes, D., and Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks, *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, 2008, pp. 319–333.

[4] Hopcroft, J. E. and Ullman, J. D.: *Introduction To Automata Theory, Languages, And Computation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[5] Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., and Teng, S.-H.: On Trip Planning Queries in Spatial Databases, *Proceedings of the 9th International Conference on Advances in Spatial and Temporal Databases*, SSTD'05, 2005, pp. 273–290.

[6] Li, J., Yang, Y., and Mamoulis, N.: Optimal Route Queries with Arbitrary Order Constraints, *IEEE Trans. on Knowl. and Data Eng.*, Vol. 25, No. 5(2013), pp. 1097–1110.

[7] Meyer, U. and Sanders, P.: Delta-Stepping: A Parallel Single Source Shortest Path Algorithm, *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, 1998, pp. 393–404.

[8] Rice, M. N. and Tsotras, V. J.: Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems, *Proceedings of the 11th International Conference on Experimental Algorithms*, SEA'12, 2012, pp. 344–355.

[9] Rice, M. N. and Tsotras, V. J.: Engineering Generalized Shortest Path Queries, *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, 2013, pp. 949–960.

[10] Serge, A., Peter, B., and Suciu, D.: *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, San Francisco, California, 2000.

[11] Sharifzadeh, M., Kolahdouzan, M., and Shahabi, C.: The Optimal Sequenced Route Query, *The VLDB Journal*, Vol. 17, No. 4(2008), pp. 765–787.

[12] Sharifzadeh, M. and Shahabi, C.: Processing Optimal Sequenced Route Queries Using Voronoi Diagrams, *Geoinformatica*, Vol. 12, No. 4(2008), pp. 411–433.

[13] Ström, N.: *Automatic continuous speech recognition with rapid speaker adaptation for human/machine interaction*, PhD Thesis, KTH, Stockholm, 1997.