ISSN 1884-0760

GRACE TECHNICAL REPORTS

Shortest Regular Category-Path Queries

Le-Duc TUNG Kento EMOTO Zhenjiang HU

GRACE-TR 2014-03 August 2014

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Shortest Regular Category-Path Queries

Le-Duc TUNG #1, Kento EMOTO *2, Zhenjiang HU #$3

# *The Graduate University for Advanced Studies*
Shonan Village, Hayama, Kanagawa 240-0193, Japan
1 `tung@nii.ac.jp`

* *Kyushu Institute of Technology*
680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan
2 `emoto@ai.kyutech.ac.jp`

$ *National Institute of Informatics*
2-1-2 Hitotsubashi, Chiyoda, Tokyo 101-8430, Japan
3 `hu@nii.ac.jp`

***Abstract*—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 (categories) of locations. The SRCP query is general and important, covering many interesting path queries such as trip planning queries, optimal sequenced route queries, optimal route queries with arbitrary order constraints. In this paper, we show that a wide class of the SRCP queries can be reduced to a series of single source shortest path searches on the input graph by using a dynamic programming formulation. As a result, we can freely exploit existing speedup techniques for single source shortest path searches to speedup the SRCP query computation. After that, we progressively engineer an efficient implementation for answering the SRCP query by using two forward and backward optimizations. Our experiments of queries on the full American road network (with over 20 million nodes and nearly 60 million edges) show that our solution is practical and scalable for large, real-world road networks.**

I. Introduction

Consider a tourist who will have a free day to travel Tokyo. A friend of hers will pick her up at her hotel, and they will enjoy a Tokyo tour ending with a dinner at the friend's home. Their tour is being planned like this: Having a breakfast at a cafeteria (`F`), then visiting two places, a museum (`M`) and a zoo (`Z`) in any order, then having lunch at a traditional Japanese restaurant (`R`), and then in the afternoon doing one of the following options; visiting a shopping mall (`S`) and then an electronics store (`E`) followed by a temple (`T`), or visiting a temple (`T`) and then a shopping mall (`S`) followed by an electronics store (`E`). In such a tour planning, they want to find a path such that (1) it visits locations of interest in a preferred order and (2) it must be the shortest path in terms of its length. This kind of query is very useful and common in practice.

A bunch of work has recently been devoted to solving various kinds of shortest path queries, where a solution path of minimum total cost must satisfy a certain constraint in terms of node categories. Li et al. [1] introduced a *trip planning query* 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 location. Rice et al. [2] proposed an exact solution for a similar problem and called it a *generalized traveling salesman path problem*. Sharifzadeh et al. [3] addressed a query called *optimal sequenced route query* in which a total order over all categories is specified, and the destination is a certain location in the last category. Li et al. [4] discussed a query that allows arbitrary (partial, total or between the two) 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. Rice et al. [5] considered a *generalized shortest path query* that is similar to the optimal sequenced route query but with a specified destination location.

While each specific class of queries is interesting, none of existing queries can directly deal with queries having multiple options as the one in the first paragraph. Constraints for locations in the morning do not require any order, so we can use a trip planning query [1]. A restaurant must be visited after a museum or a zoo, which is a query with arbitrary order constraints [4]. Two options in the afternoon are two queries with total order constraints [3], but they cannot be expressed by a query with arbitrary order constraints [4] because these options expose the contraint "a temple (`T`) is prohibited only in between a shopping mall (`S`) and an electronics store (`E`)" that cannot be expressed by any partial order.

To remedy such above queries, we propose a more general class of queries, called *shortest regular category-path (SRCP) queries*, where path constraints on the node categories are described by a regular category-path expression. The SRCP queries are powerful, covering all the existing queries discussed above as well as their combinations. Recall the tour example above, we can describe it as an SRCP query as a triple:

$$\langle \mathtt{h}, \mathtt{o}, \mathtt{F}(\mathtt{MZ} \mid \mathtt{ZM})\mathtt{R}(\mathtt{SET} \mid \mathtt{TSE})\rangle$$

where the first component specifies the source (the hotel), the second component specifies the target (the home), and the third component is a regular category-path expression describing the constraints on paths. Suppose that, in the late afternoon, we could walk around some libraries (`L`) or cafeterias (`F`) before going home. Then we can describe it with the following naive query:

$$\langle \mathtt{h}, \mathtt{o}, \mathtt{F(MZ\,|\,ZM)R(SET\,|\,TSE)(L|F)}^{+} \rangle$$

Now, the challenge is how to solve the SRCP queries efficiently. One direct idea is to utilize the existing solution proposed by Barret et al. [6] on formal-language-constrained shortest path query, in which shortest paths are constrained by a formal language (including regular expressions) for describing requirements on labels associated with nodes/edges of an input graph. Barret's algorithm computes a product graph of the input graph and a nondeterministic finite automaton constructed from the regular expression, and reduces the query computation to a point-to-point shortest path search on the product graph. However, two problems remain in this approach. First, the product graph could be very huge, and the algorithm would require searching many (or even all) nodes in the product graph. Second, it would be impractical to utilize existing effective preprocessing techniques (e.g., Contraction Hierarchies [7], Precomputed Cluster Distance [8], etc.) for improving the point-to-point shortest path searches, because we have to generate a product graph for each query and thus the preprocessed product graph cannot be used for another query.

In this paper, we show that a wide practical class of the SRCP queries can be reduced to a series of single source shortest path searches on the input graph by using a dynamic programming formulation; thus, we can exploit existing speedup techniques for single source shortest path searches freely to obtain efficient implementations for answering the SRCP queries.

Our main contributions are summarized as follows.

- We propose a *general mechanism* (SRCP queries) for description of various complex kinds of shortest path queries with path constraints on node categories. It covers the existing queries such as trip planning queries, optimal sequenced route queries, optimal route queries with arbitrary order constraints, generalized traveling salesman path problem, and generalized shortest path queries.
- We show that a wide practical class of SRCP queries can be *efficiently computed*, by reducing an SRCP query to a series of single source shortest path searches, and establishing a dynamic programming formulation for the SRCP query computation. In addition, our approach enables utilization of any fast single source shortest path search (sequential or parallel search) to gain the best performance.
- We progressively engineer an efficient implementation for answering the SRCP queries by using two forward and backward optimizations. Our experiments of queries on the full American road network (with over 20 million nodes and nearly 60 million edges) show that our solution is practical and scalable for large, real-world road networks.

The rest of this paper is organized as follows. Section II gives a formal definition of the Shortest Regular Category-Path query, and discusses its expressiveness. We first introduce a basic solution based on dynamic programming for answering the SRCP queries in Sect. III and then we engineer it to get an efficient algorithm in Sect. IV. A set of experiments with large data sets is showed in Sect. V. Section VI discusses some related works and Sect. VII concludes the paper.

II. Shortest Regular Category-Path Queries

In this section, we formally define the shortest regular category-path (SRCP) queries and the related SRCP problem, and demonstrate the expressiveness of SRCP queries through several typical examples.

A. SRCP Queries

The SRCP query is a special case of the known shortest path query.

Definition II.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 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 $d(s,t)$.*

The SRCP query is defined over node categories.

Definition II.2. *(Category) Given a weighted digraph $G = (V, E, w)$. A category $\mathtt{C}$ is a set of nodes in G, or $\mathtt{C} \subseteq V$.*

Definition II.3. *(Regular Expression on Category) The syntax for regular expression on Category is:*

$$R ::= \hat{\mathtt{C}} \mid RR \mid R \text{ “}|\text{” } R \mid R\text{“}*\text{”}$$

Here $\hat{\mathtt{C}}$ is to recognize a category $\mathtt{C}$, i.e., a terminal symbol, R_1R_2 denotes the concatenation, $R_1 \,|\, R_2$ denotes the alternation, and $R*$ denotes closure (Kleene star). As usual, we may write R^+ for $RR*$.

Definition II.4. *(Path Satisfaction) A path $P_{s,t} = \langle v_1, v_2, \ldots, v_k \rangle$ from s to t is said to satisfy a regular expression R over a set of categories if the concatenation of categories of these nodes spells out R. Such a path is denoted by $P_{s,R,t}$.*

Definition II.5. *(Shortest Regular Category-Path (SRCP) / 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 a regular expression over C_is. An SRCP query is represented as a triple*

$$\langle s, t, R \rangle$$

where s and t denote the starting and ending nodes respectively.

A path $P^{min}_{s,R,t}$ is called an SRCP if it satisfies 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 II.6. *(SRCP problem) An SRCP problem is to find an SRCP for a given SRCP query.*

B. Simplification of SRCP Queries

In general, SRCP queries are more difficult to solve than existing category-constrained shortest path queries. The difficulty comes from two constructors in the SRCP queries, one is the closure and the other is the alternation. To provide an efficient and practical algorithm to solve SRCP queries, we simplify them by restricting the use of the closure.

Given an SRCP query

$$\langle s, t, R\rangle$$

where R is a regular expression, we show that we can simplify the SRCP problem a lot by localizing the global closure. This is based on the following two observations. First, it is usually more practical to consider a path passing a node of category $\mathtt{C}$ (i.e., $\_*\hat{\mathtt{C}}\_*$, where _ denotes an arbitrary category) rather than a path *just* containing an *exactly single node* of category (i.e., $\hat{\mathtt{C}}$). This would suggest us to treat $\_*\hat{\mathtt{C}}\_*$ as a primitive. Second, concatenation of $\_*\hat{\mathtt{C}}\_*$ with a closure will cancel the effect of the closure. This means that the following two SRCP queries,

$$\langle s, t, \_*\hat{\mathtt{C}}\_*R*\rangle$$
$$\langle s, t, R*\_*\hat{\mathtt{C}}\_*\rangle$$

will have the same effect as the query

$$\langle s, t, \_*\hat{\mathtt{C}}\_*\rangle$$

It is because the shortest path obtained from the last query should be the shortest path from the first two queries.

Given the above, we will simplify regular expressions to make the closure appear only in the form of $\_*\hat{\mathtt{C}}\_*$.

Definition II.7. *(Simplified Regular Expression (SRE)) The syntax for SREs is:*

$$R ::= \_*\hat{\mathtt{C}}\_* \mid RR \mid R \text{ “}|\text{” } R$$

For simplicity, we use “C” as an abbreviation of “$\_*\hat{\mathtt{C}}\_*$”.

In the rest of this paper, we will focus on SRCP queries where simplified regular expressions are considered.

Figure 1 shows an example SRCP query and paths satisfying the query $\langle s, t, \mathtt{O}\,(\mathtt{GY}\,|\,\mathtt{B})\,\mathtt{V}\rangle$. 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 $\langle s, o_1, g_2, y_0, \ldots, v_0, \ldots, t\rangle$ spells out the constraint “OGYV”, however the node y_0 in Y connects to the node v_0 in V via two other nodes, although Y and V are contiguous in the constraint.

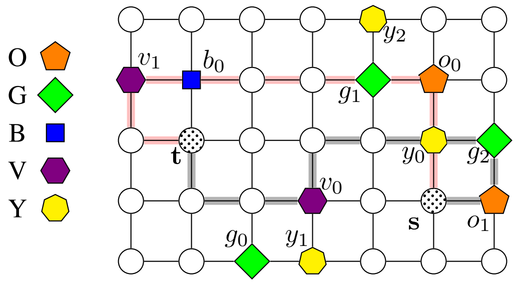


Fig. 1: An SRCP query example $\langle s, t, \mathtt{O}\,(\mathtt{GY}\,|\,\mathtt{B})\,\mathtt{V}\rangle$. All edges have the same weight of 1. Two of the SRCPs are shaded in grey and pink. $d^R(s,t) = 9$

C. Expressiveness

We show that SRCP queries are powerful enough to express many interesting category-constrained shortest path queries including those with partial or total order constraints.

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 [5]. A GSP is expressed in our SRCP query as follows.

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

where s and t are the starting point and destination point, respectively.

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 [3]. This query is different from 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 weight 0 from nodes in the last category of the order constraints to t'. The SRCP query is then as follows.

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

3) Trip Planning Queries/Generalized Traveling Salesman Path Problem Queries (TPQ/GTSPP): A trip planning query [1] or generalized traveling salesman path problem query [2] 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.

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

where R_is ($i = 1\ldots k!$) are permutations of the set $\{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_k\}$. For example, R_1 is $\mathtt{C}_1\mathtt{C}_2\ldots\mathtt{C}_k$, R_2 is $\mathtt{C}_2\mathtt{C}_1\ldots\mathtt{C}_k$, and so on.

4) Optimal Route Queries (ORQ) 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}) [4].

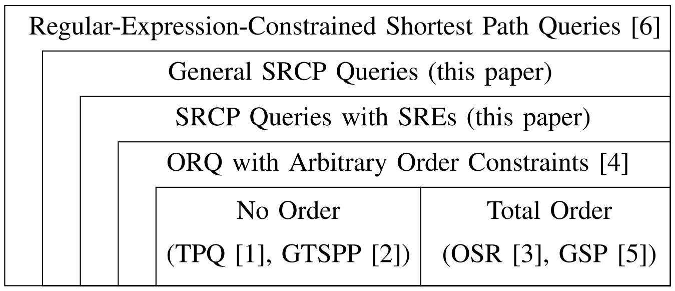

Regular-Expression-Constrained Shortest Path Queries [6]
General SRCP Queries (this paper)
SRCP Queries with SREs (this paper)
ORQ with Arbitrary Order Constraints [4]
No Order
(TPQ [1], GTSPP [2])
Total Order
(OSR [3], GSP [5])

Fig. 2: The relationship between SRCP queries and existing queries

However, here users can specify partial 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 node 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 needed to be done. First, we need generate total order constraints from the visit order graph, then put them together in the form of SRCP queries by using alternation operators. 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 create an artificial destination node t' for the SRCP query, which is done by adding edges with weight 0 from nodes in all categories in the set of categories to t'. The SRCP query is finally as follows.

$$\langle s, t', R_1 \,|\, R_2 \,|\, \ldots \,|\, R_l \rangle$$

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

In summary, Figure 2 shows the relationships between our SRCP queries and other queries. The general SRCP query is a subset of the regular-expression-constrained shortest path query [6] in which its regular expression is defined over node labels. A practical subset of the general SRCP queries, whose constraints are simplified regular expressions, is considered in this paper. Although it is limited but covers all existing important problems such as queries with arbitrary order, total order, or no order constraints.

III. SRCP Query Algorithm

A naive approach for answering the SRCP query (s, t, R) is considering it as a combination of existing queries with total order constraints. To do that, we first generate all total order constraints R_i of the input SRE R. For example, consider an SRE "$\mathtt{B(C\,|\,S)}$", we generate two equivalent constraints R_1 = "$\mathtt{BC}$" and R_2 = "$\mathtt{BS}$". Then we evaluate sub-queries $\langle s, t, R_i \rangle$, e.g. $\langle s, t, \mathtt{BS} \rangle$, $\langle s, t, \mathtt{BC} \rangle$, independently by efficient algorithms for total order constraints (e.g. algorithms for optimal sequenced route [3] or generalized shortest path [5]). Finally we take the minimum cost from results of each sub-queries. This approach is straightforward but inefficient due to many redundant computations between the evaluations of sub-queries, e.g. two sub-queries in the above example would share a computation for paths from $\mathtt{s}$ to the category $\mathtt{B}$. Moreover, computations from the category $\mathtt{B}$ to the category $\mathtt{C}$ and $\mathtt{S}$ can be overlapped in terms of visited edges/nodes.

In this section, we propose a dynamic programming solution to solve the SRCP problem in a more efficient way, in which we reduce the SRCP problem to a series of single source shortest path searches.

A. SREs as Directed Acyclic Graphs

It is well known that a regular expression can be expressed by a nondeterministic finite state automaton with ϵ-transitions (NFA-ϵ) [9]. However, the use of ϵ-transitions is not necessary due to the absence of closures in SREs of SRCP queries. Hence, we directly transform an SRE to an NFA without ϵ-transitions. This NFA is a directed acyclic graph (DAG) G_R that represents the structure of SREs in SRCP queries. Nodes of G_R are object identifiers (OIDs) that are integers uniquely identifying a node, and edges of G_R are labeled by categories in R.

The graph G_R is simply constructed by a recursive function on the structures of SREs. Algorithm 1 is to generate a DAG graph for an given SRE. Given an SRE R, the recursive function $\mathit{ParseSRE}$ computes a triple $\langle sc, G_R, sk \rangle$ where sc and sk are the source and sink node in the graph G_R, respectively. For the terminal case $R = \mathtt{C}$ (see Alg. 1, lines 2-6), we create a singleton graph G_C containing only one edge d labeled by the category $\mathtt{C}$. The source and sink node of G_C are respectively the source and target node of d (see Fig. 3a for an illustration of G_C). For the concatenation case $R = R_1 R_2$ (see Alg. 1, lines 7-13), the graph G_R is constructed from two graphs G_{R_1} of R_1 and G_{R_2} of R_2 by merging the sink node of G_{R_1} with the source node of G_{R_2} (see Fig. 3b for an example). For the alternation case $R = R_1 \,|\, R_2$ (see Alg. 1, lines 14-22), we create the graph G_R from G_{R_1} and G_{R_2} by merging the source node of G_{R_1} with the source node of G_{R_2}, and the sink node of G_{R_1} with the sink node of G_{R_2} (see Fig. 3c for an example). The generated graph G_R is a DAG with one source node and one sink node.

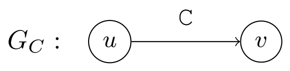
To present the semantics of the whole query $Q = \langle s, t, R \rangle$, we introduce a *query graph* $G_Q = (V_Q, E_Q)$ that is constructed from the graph G_R by attaching an in-coming edge labeled by $\{s\}$ to the source node of G_R, and an out-going edge labeled by $\{t\}$ to the sink node of G_R. Figure 4 shows a query graph of the query $\langle s, t, \mathtt{O\,(GY\,|\,B)\,V} \rangle$, in which two sets $\{s\}$ and $\{t\}$ are called dummy categories (superscripts of edge labels will be explained later in the Sect. III-B).

B. Dynamic Programming Formulation

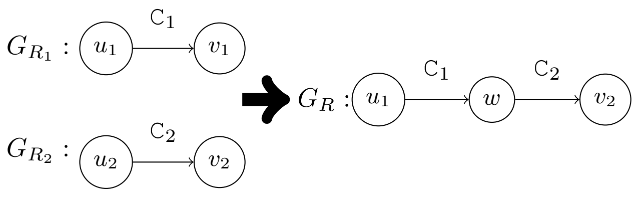
Next, we formalize a dynamic programming formulation for the SRCP problem by using the query graph. Given an SRCP query $Q = \langle s, t, R \rangle$, we establish a DP table X in order to store values during computation. Each row in the

Algorithm 1: ParseSRE(R)

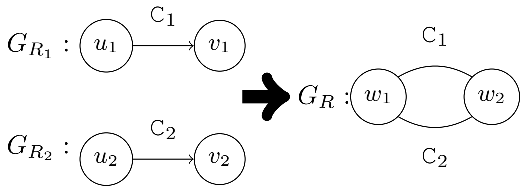
```
input   : A simplified regular expression $R$
output  : A triple $\langle sc, G_R, sk\rangle$
1:  switch $R$ do
2:      case $C$
3:          $u \leftarrow new\ Vertex()$;
4:          $v \leftarrow new\ Vertex()$;
5:          $G_C \leftarrow new\ DAG(\{u, v\}, \{(u, C, v)\})$;
6:          return $\langle u, G_C, v\rangle$
7:      case $R_1R_2$
8:          $\langle sc_1, G_{R_1}, sk_1\rangle \leftarrow ParseSRE(R_1)$;
9:          $\langle sc_2, G_{R_2}, sk_2\rangle \leftarrow ParseSRE(R_2)$;
10:         $w \leftarrow new\ Vertex()$;
11:         replace $sk_1 \in G_{R_1}$ and $sr_2 \in G_{R_2}$ by $w$;
            /* merge two graphs */
12:         $G_R \leftarrow new\ DAG(V(G_{R_1}) \cup V(G_{R_2}), E(G_{R_1}) \cup E(G_{R_2}))$;
13:         return $\langle sc_1, G_R, sk_2\rangle$
14:     case $R_1 \mid R_2$
15:         $\langle sc_1, G_{R_1}, sk_1\rangle \leftarrow ParseSRE(R_1)$;
16:         $\langle sc_2, G_{R_2}, sk_2\rangle \leftarrow ParseSRE(R_2)$;
17:         $sc_{12} \leftarrow new\ Vertex()$;
18:         $sk_{12} \leftarrow new\ Vertex()$;
19:         replace $sc_1 \in G_{R_1}$ and $sc_2 \in G_{R_2}$ by $sc_{12}$;
20:         replace $sk_1 \in G_{R_1}$ and $sk_2 \in G_{R_2}$ by $sk_{12}$;
            /* merge two graphs */
21:         $G_R \leftarrow new\ DAG(V(G_{R_1}) \cup V(G_{R_2}), E(G_{R_1}) \cup E(G_{R_2}))$;
22:         return $\langle sc_{12}, G_R, sk_{12}\rangle$
23: endsw
```



(a) Singleton graph G_C



(b) Concatenation case: $R = R_1R_2$ (e.g. $R_1 = C_1$, $R_2 = C_2$)



(c) Alternation case: $R = R_1 \mid R_2$ (e.g. $R_1 = C_1$, $R_2 = C_2$)

Fig. 3: Query graph constructors

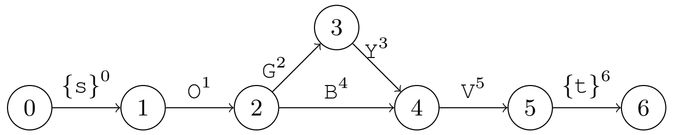


Fig. 4: A query graph of the query $\langle s, t, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$. Integers in nodes are OIDs. Superscripts of categories are equivalent row indices in the DP table. The subgraph from the node 1 to the node 5 is a DAG graph corresponding to the SRE $\mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}$

table corresponds to a category on an edge of the graph G_Q. Therefore, the table X has $|E_Q|$ rows, and g columns where g is the maximum size of categories in the query Q. "$X[i, j]$" 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 $\langle s, t, \mathtt{O}\,(\mathtt{GY} \mid \mathtt{B})\,\mathtt{V}\rangle$, we have $\langle s^0, t^6, \mathtt{O}^1\,(\mathtt{G}^2\mathtt{Y}^3 \mid \mathtt{B}^4)\,\mathtt{V}^5\rangle$.

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

$$\begin{aligned} in_u &= \{r | (v, u) \in E_Q, C^r \leftarrow l(v, u)\} \\ out_u &= \{r | (u, w) \in E_Q, C^r \leftarrow l(u, w)\}, \end{aligned}$$

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 an edge (u, v) in G_Q. Computation steps form a dynamic programing formulation for the SRCP problem. Following is the computing formulation of the computation step $in_u \rightarrow out_u$.

$$\underset{i \in out_u}{X[i, j]} = \begin{cases} 0 & \text{If } i = 0 \\ \min\limits_{r \in in_u} \{ \min\limits_{0 \le l < |C^r|} \{X[r, l] + d(c_{r,l}, c_{i,j})\}\} & \text{If } i > 0 \end{cases}$$

where, C^r is the category corresponding to the r-th row in the DP table X, $c_{i,j}$ is the j-th node in the category C^i.

Lemma III.1. *Value $X[i, j]$ in the DP table represents the optimal cost of the SRCP of the query $\langle s, c_{i,j}, R^i\rangle$ where R^i is the SRE corresponding to a subgraph of G_Q that includes all paths from the node just after the source node of G_Q to the source node of the edge labeled C^i.*

For example, consider the query $\langle s^0, t^6, O^1\,(G^2Y^3 \mid B^4)\,V^5\rangle$, its query graph 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 $R^3 = O^1G^2$ corresponding to the subgraph having edges from the node 1 (the node just after the source node of G_Q) to the node 3 (the source node 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 nodes in G_Q).

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

For the induction step, $k > 1$. Let u be the node in G_Q that generates the k-th computation step. For a set of nodes V_u that are before the node u in the topological sort of G_Q, let E_u be a set of edges related to nodes 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 node v in G_Q, that is $in_v \to out_v$, in which in_v and out_v is the set of row indices of all in-coming and out-going edges of the node v, respectively. It is clear that $in_v \subseteq ps$. Since each $c_{i,j}$ is a member of category C^i, $i \in out_v, j = 0 \ldots (|C^i| - 1)$, it suffices to find the shortest path cost from nodes $c_{r,l}$ in categories C^r ($r \in in_v, 0 \le l < |C^r|$) to $c_{i,j}$. It follows from our induction hypothesis that the value of $X[i,j]$ is computed by $\min\limits_{r \in in_v} \{ \min\limits_{0 \le l < |C^r|} \{X[r,l] + d(c_{r,l}, c_{i,j})\}\}$. ∎

Corollary III.2. *Let m be the row index of the dummy category $\{t\}$. Value $X[m,0]$ represents the cost of the SRCP of the query $\langle s, t, R \rangle$.*

Figure 5 shows a table containing costs during the computation of the query $\langle s^0, t^6, \mathtt{O}^1\,(\mathtt{G}^2\mathtt{Y}^3 \,|\, \mathtt{B}^4)\,\mathtt{V}^5 \rangle$. The order of computation steps is as follows.

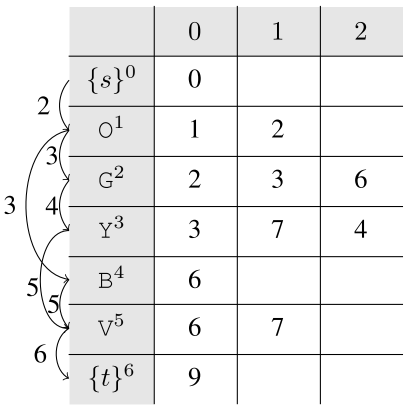
1st step:	$[]$	$\to [0]$
2nd step:	$[0]$	$\to [1]$
3rd step:	$[1]$	$\to [2,4]$
4th step:	$[2]$	$\to [3]$
5th step:	$[3,4]$	$\to [5]$
6th step:	$[5]$	$\to [6]$
7th step:	$[6]$	$\to []$

The optimal cost $d^R(s,t) = 9$ of the query is stored at the last row (its row index is 6 that is the index of the dummy category $\{t\}$) of the table. Note that the first step is actually to initialize the value of $X[0,0]$, and the last step does nothing.

C. Single Source Shortest Path (SSSP) Search

A computation step "$xs \to ys$" 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 "$xs \to ys$" 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 DP table [5], 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 III.3. *Given a weighted digraph $G = (V, E, w)$ and an SRCP query $Q = \langle s, t, R \rangle$. Let $G_Q = (V_Q, E_Q)$*



	0	1	2
$\{s\}^0$	0		
$\mathtt{O}^1$	1	2	
$\mathtt{G}^2$	2	3	6
$\mathtt{Y}^3$	3	7	4
$\mathtt{B}^4$	6		
$\mathtt{V}^5$	6	7	
$\{t\}^6$	9		

Fig. 5: A table of optimal costs for the query in Fig. 1. The first column contains names of categories. The first row contains indices of nodes in a category. Each row contains optimal costs from s to a node in a category via some other categories according to the SRE. Curved arrows on the left indicate computation steps and its orders.

be the query graph of Q and T_{sssp} be the complexity of a single source shortest path search, the cost of our algorithm is $O(|V_Q| T_{sssp})$, and the space complexity of the algorithm is $O(|E_Q|)$.

Corollary III.4. *Given a weighted digraph $G = (V, E, w)$ and an SRCP query $Q = \langle s, t, R \rangle$. Let $G_Q = (V_Q, E_Q)$ be the query graph of Q. When a Dijkstra algorithm with a Fibonacci heap is used for SSSP searches [10], our algorithm answers Q in the time complexity of $O(|V_Q|(|E| + |V| log |V|))$.*

One advantage of this approach is that it is independent of the underlying SSSP search. Thus, we can use any fast SSSP algorithm to implement, such as Contraction Hierarchies technique [7], [5], Delta-Stepping [11], PHASE [12], etc. This is useful because we can apply different efficient fast SSSP algorithms for different kinds of graphs (road networks, social networks, biological networks, etc.)

IV. Optimizations

Although the dynamic programming formulation can help solve the SRCP problem, there is a need in optimizing the SRCP query algorithm. First, the number of computation steps ($|V_Q|$) depends on user-defined queries. For example, two queries, $\langle s, t, (\mathtt{OGYV} \,|\, \mathtt{OBV}) \rangle$ and $\langle s, t, \mathtt{O}\,(\mathtt{GY} \,|\, \mathtt{B})\,\mathtt{V} \rangle$, 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 $\langle s, t, R_1 \,|\, R_2 \,|\, \ldots \,|\, R_{k!} \rangle$ where R_is are permutations of the set $\mathtt{C}$, $i = 1 \ldots k!$. In this case, $|V_Q|$ will be $((k-1)k! + 2)$ which is not practical even for small k (e.g. $k = 5$. See Sect. V for more discussion). This section will discuss how to engineer an efficient algorithm for answering the SRCP query.

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 nodes and the number of edges. The number of nodes affects the time complexity and the number of edges affects the space complexity that 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 nodes that generates exactly the same total order constraints as a given query graph.

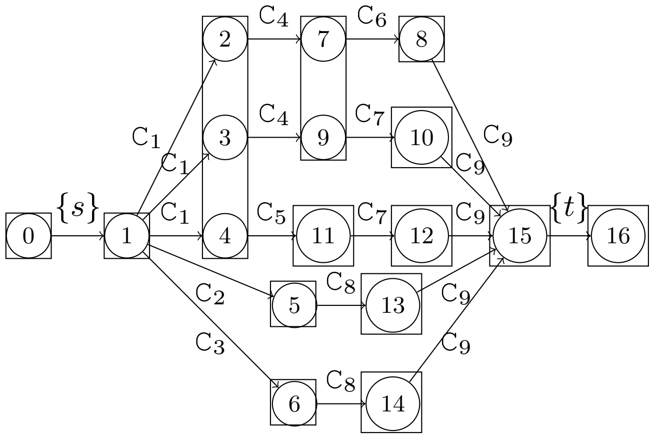
The query graph is a subclass of non-deterministic finite state automatons (NFA) [9]. It is well known that NFA minimization is computationally hard, and cannot in general be solved in polynomial time. There exists a well-known algorithm for minimizing NFAs [9]. The algorithm computes a minimal equivalent deterministic finite automaton (DFA) with respect to the number of nodes, and consists of two steps: determinization and minimization. The determinization step is to compute a DFA from an NFA, then the DFA is minimized by the minimization step 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 that represent a set of hypotheses in speed recognition system. A word graph is a DAG with exactly one source node and one sink node. In this section we apply these algorithms to optimize the query graph which has a similar structure to word graphs.

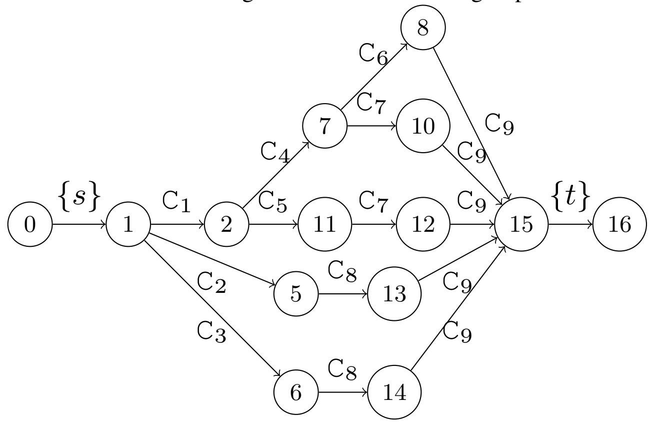
A. Forward Optimization (Determinization)

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 a determinization algorithm is identifying the correspondence between a set of nodes in the original graph and a node in its DFA graph. This leads to an exponential computation because there are 2^N sets of nodes in a graph of N nodes. However, the algorithm can be simplified in the case of the query graph that is a DAG with one source node and one sink node.

The idea of a forward optimization is scanning the query graph G_Q from its source node to its sink node, and grouping nodes that come from the same node with the same categories [13]. Let DG_Q be a deterministic query graph that is the result of the forward optimization, a node in DG_Q is correspondent to a set of nodes in G_Q. Figure 6 describes an example of the forward optimization for the query $\langle s, t, \mathtt{C}_1\mathtt{C}_4\mathtt{C}_6\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_4\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_5\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_2\mathtt{C}_8\mathtt{C}_9 \mid \mathtt{C}_3\mathtt{C}_8\mathtt{C}_9\rangle$. Initially, the graph DG_Q contains only one node created from the source node 0 of G_Q. Because there is only one edge going out from the node 0 of G_Q, a new node in DG_Q is also correspondent to only one node in G_Q (node 1). In the next step, since there are three out-going edges of the node 1 in G_Q with the same category $\mathtt{C}_1$, we can group their target nodes into one set $\{2, 3, 4\}$ and create a new equivalent node in DG_Q. Two nodes 7 and 9 in G_Q are both coming from nodes in the same set $\{2, 3, 4\}$ with the same categories $\mathtt{C}_4$, thus we also group them to create a new node in DG_Q, This procedure is performed until reaching the sink node of G_Q.



(a) A query graph G_Q of the query $\langle \mathtt{s}, \mathtt{t}, \mathtt{C}_1\mathtt{C}_4\mathtt{C}_6\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_4\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_1\mathtt{C}_5\mathtt{C}_7\mathtt{C}_9 \mid \mathtt{C}_2\mathtt{C}_8\mathtt{C}_9 \mid \mathtt{C}_3\mathtt{C}_8\mathtt{C}_9\rangle$. Nodes in the same rectangular are needed to be grouped



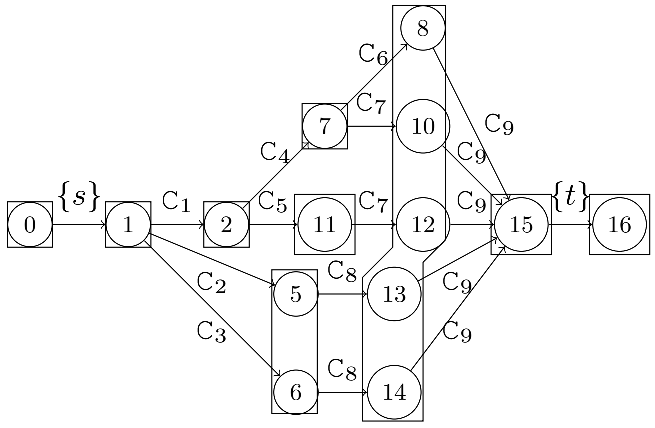
(b) A deterministic query graph DG_Q of G_Q

Fig. 6: Forward optimization.

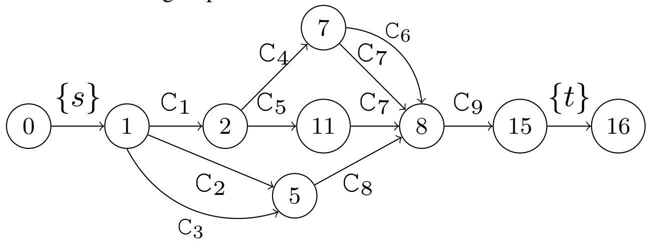
B. Backward Optimization (Minimization)

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

Figure 7 describes a process of the backward optimization. We initialize the minimal deterministic query graph MDG_Q with the sink node in the input deterministic query graph DG_Q. Then we scan the graph DG_Q from the sink node back to the source node in order to add new equivalent nodes to the graph MDG_Q. For each pair of nodes, the information we need to compare is the categories on the out-going edges



(a) A deterministic query graph DG_Q. Nodes in the same rectangular are needed to be grouped



(b) Minimal deterministic query graph MDG_Q of DG_Q

Fig. 7: Backward optimization

and the nodes that they go to. To store such information, for each node, we maintain a set of tuples of an out-going edge's label and an equivalent target node. If there are multiple nodes having the same set, then we group them to create a new node in the graph MDG_Q. For example in Fig. 7, because five nodes $8, 9, 10, 12, 13, 14$ in MD_Q (see Fig. 7a) go to the same node 15 with the same edge label $\mathtt{C}_9$, so we group them and create an equivalent node 8 in the graph MDG_Q (see Fig. 7b). Note that, although two nodes 7 and 11 in DG_Q go to nodes in the same set $\{8, 9, 10, 12, 13, 14\}$, we do not group them. This is because the node 7 has two edges to nodes in $\{8, 9, 10, 12, 13, 14\}$, while the node 11 has only one edge to nodes in $\{8, 9, 10, 12, 13, 14\}$.

C. *Optimization of the DP Table*

For our approach, the number of rows in the DP table is equal to the number of edges in the query graph G_Q. Although the optimization of the number of nodes in G_Q also causes a decrease of the number of edges, this decrease is not remarkable. Moreover, because the number of elements in each row of the table is equal to 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.

One solution to managing the DP table is dynamically creating it. 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 out-going edges, and never used again. Thus, after each computation step, we need not store rows corresponding to in-coming edges.

D. *Optimization of the Query Structure*

Although two forward and backward optimizations result in a minimal query graph G_Q in terms of the number of nodes, for some user-defined queries, we can get a smaller graph G_Q by preprocessing the SREs in the user-defined queries.

Consider SRCP queries in the following form

$$\langle s, t, R_1 R_2 R_3 \mid R_2 \rangle$$

where R_1, R_2, R_3 are arbitrary SREs. They have the same effect as the query

$$\langle s, t, R_2 \rangle$$

Therefore, for this kind of queries, we can eliminate all SREs that contain another SREs. This can be generalized for SRCP queries having more alternatives.

V. Implementation and Evaluation

We implemented a framework[1] to answer the SRCP queries. The overview of our framework is showed in the Fig. 8. First, it takes an SRCP query as input, parses it to get a query graph, then optimizes the query graph by two steps "determinization" and "minimization". Next, it will generate a sequence of computation steps, in which 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 [5], 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 on road networks [14]. Its idea is preprocessing a graph by augmenting it with shortcuts so that shortest path costs are preserved. Shortcuts are then intensively used by a bidirectional Dijkstra algorithm to speed up the shortest path search on the augmented graph.

A. *Environment*

All our experiments were ran 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 create and access an augmented graph.

Experiments were performed with a graph of the Full USA road network, having $23,947,347$ nodes and $58,333,344$ edges. We borrowed the graph from the benchmarks of the

[1]http://www.prg.nii.ac.jp/members/tungld/srcp-July2014.tar.gz
[2]http://algo2.iti.kit.edu/source/contraction-hierarchies-20090221.tgz

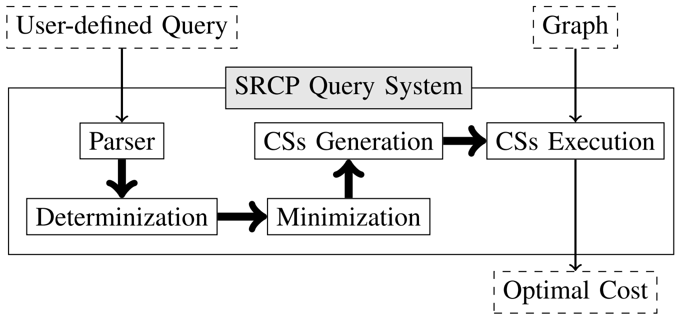


Fig. 8: The overview of our framework

9th DIMACS implementation challenge [3]. It took about 25 minutes to create an augmented graph using the CH technique (this graph will be used as a common input graph for programs in our experiments)

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

- **gsp**: the algorithm proposed by Rice [5] to answer the Generalized Shortest Path Query that uses total order constraints $\langle s, t, \texttt{C}_1\texttt{C}_2 \dots \texttt{C}_k\rangle$. We implemented the core of this algorithm (without some heuristic technique).
- **perm**: a naive algorithm, which was mentioned in the beginning of the Sect. III, 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 evaluated by the **gsp** algorithm.
- **srcp-noopt**: our algorithm for the SRCP query without optimizations.
- **srcp-opt**: our algorithm for the SRCP query with optimizations. Two optimizations were implemented: the forward and the backward. To store a set of nodes, we used a standard class *std::set* which supports *equality comparison*. We used a class *std::unordered_map* as a hash table to store sets of nodes, which allows for fast access to the sets of nodes to determine their existing and elements. The boost graph library [4] is used to implement the optimizations.

Categories used in our queries are generated randomly and have the same number of nodes.

B. Results

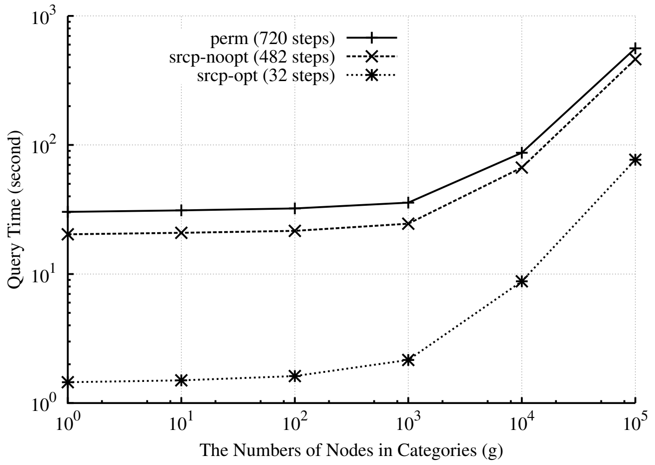
1) Influence of the Optimizations: We measure the performance of our algorithm for the trip planning queries which are expensive queries. A Trip Planning Query (TPQ) with k categories is written in the SRCP queries as follows.

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

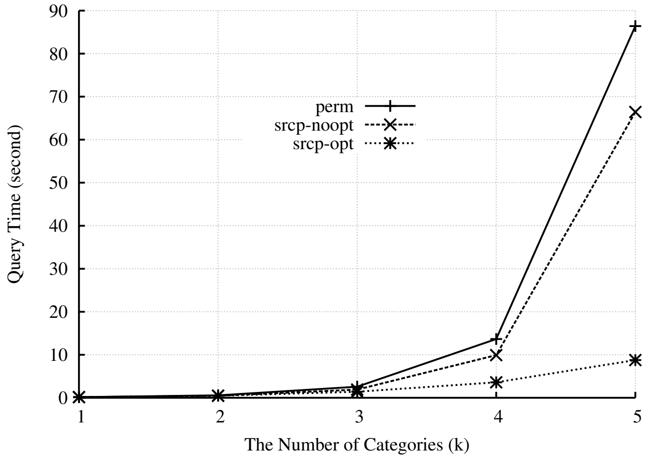
where R_is are permutations of the set $\{\texttt{C}_1, \texttt{C}_2, \dots, \texttt{C}_k\}$, $i = 1 \dots k!$. For example, R_1 is $\texttt{C}_1\texttt{C}_2 \dots \texttt{C}_k$, R_2 is $\texttt{C}_2\texttt{C}_1 \dots \texttt{C}_k$, and so on.

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

[4]http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/



(a) TPQ/GTSPP queries. Vary the size of categories (g), fix the number of categories ($k = 5$)



(b) TPQ/GTSPP queries. Vary the number of categories (k), fix the size of categories ($g = 10,000$)

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 computation steps ($(5 + 1) * 5!$). Without optimizations, 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. 9a. It shows that the **srcp-opt** 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 in Fig. 9b, the running time of the **perm** algorithm is significantly increased, while the **srcp-opt** algorithm is quite stable. Although the **srcp-noopt** algorithm can reduce the number of computation steps twice, it still follows an factorial running time. This experiments show the important role of optimizations in our solution.

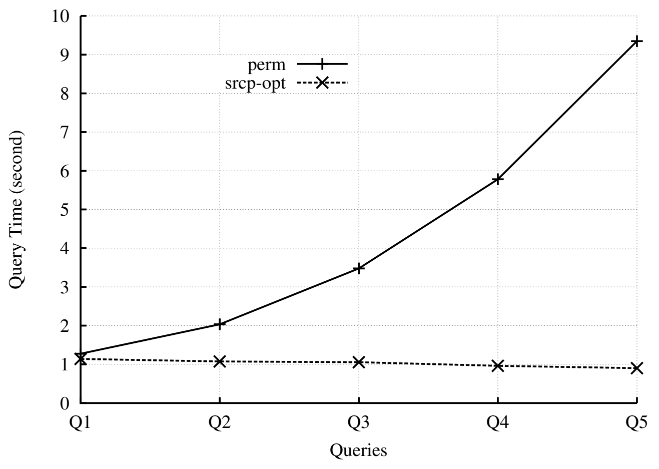


Fig. 10: Queries with multiple options ($k = 8, g = 10,000$)

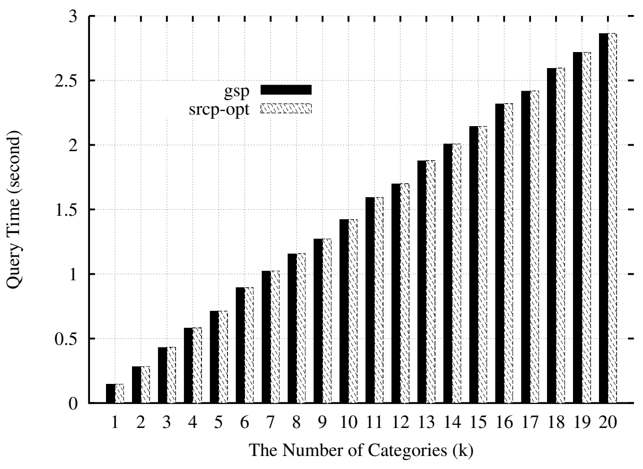


Fig. 11: GSP queries ($g = 10,000$)

TABLE I: The number of computation steps in each query

	Q1	Q2	Q3	Q4	Q5
perm	9	16	28	48	80
srcp-opt	9	8	7	6	5

TABLE II: Performance of optimizations

	1	2	3	4	5	6
forward (ms)	0.026	0.033	0.069	0.290	2.566	53.697
backward (ms)	0.018	0.034	0.084	0.302	1.580	10.872

2) Influence of the Alternation Operators: To see the impact of alternation operators ($|$) for a given query on the performance of our optimal algorithm, we do experiments with queries which differ in the number of alternation operators, while the number of categories is the same. Starting with a query without alternatives, each time we insert one "$|$" operator to create a new query. In particular, we use the following queries.

$$Q1 = \langle s, t, \mathtt{C}_1\mathtt{C}_2\mathtt{C}_3\mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8\rangle$$
$$Q2 = \langle s, t, (\mathtt{C}_1 \mid \mathtt{C}_2)\mathtt{C}_3\mathtt{C}_4\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8\rangle$$
$$Q3 = \langle s, t, (\mathtt{C}_1 \mid \mathtt{C}_2)(\mathtt{C}_3 \mid \mathtt{C}_4)\mathtt{C}_5\mathtt{C}_6\mathtt{C}_7\mathtt{C}_8\rangle$$
$$Q4 = \langle s, t, (\mathtt{C}_1 \mid \mathtt{C}_2)(\mathtt{C}_3 \mid \mathtt{C}_4)(\mathtt{C}_5 \mid \mathtt{C}_6)\mathtt{C}_7\mathtt{C}_8\rangle$$
$$Q5 = \langle s, t, (\mathtt{C}_1 \mid \mathtt{C}_2)(\mathtt{C}_3 \mid \mathtt{C}_4)(\mathtt{C}_5 \mid \mathtt{C}_6)(\mathtt{C}_7 \mid \mathtt{C}_8)\rangle$$

Figure 10 shows the result. It is interesting that when there are more options in the SRCP query, our algorithm becomes faster. Looking the Table I that shows the number of computation steps for each query, we see that the reason of such good performance is that the number of computation steps is reduced, 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 dramatically increased because there are many total order constraints generated. This result also shows that our algorithm can reduce a large amount of redundant computation steps when alternation operators appear in the query.

3) Overhead of Optimizations: First, we measure the performance of our algorithm when answering the query GSP. We compare it to the algorithm proposed by Rice [5]. As can be seen in the Fig. 11, two algorithms have the same performance when the number of categories is increased. 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.

However, for the trip planning queries, the overhead of optimizations, in particular, the overhead of the forward optimization, is expensive. Table II shows the running times of the forward and backward optimizations when the number of categories (k) is changed from 1 to 6. With small k $(1, 2, 3, 4)$, the backward optimization takes less time than the forward optimization. Nevertheless, the forward optimization is more expensive with larger k $(5, 6)$. This is because, the forward optimization takes an exponential time complexity, while the backward optimization takes a linearithmic time complexity. Furthermore, both optimizations highly depend on the way of storing sets of nodes during their computations. This effects the performance of determining whether a set already exists or not.

VI. Related Work

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 [1] 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 with respect to the number of categories. 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. [2] 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 (CH) [7], resulting in the time complexity of $O(2^k(m' + |V|k)$, in which m' is the number of edges of the preprocessed graph. It is noted that the improved algorithm is slightly different from the orignal algorithm, in which it executes a series of sweeping phases according to levels of an abstraction of the product graph, and highly depends on the CH technique.

In parallel to Li et al.'s work [1], Sharifzadeh et al [3] 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 starting from a given point and passing through a number of categories in a particular order. They proposed two 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. [15] 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. [5] 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 but having only one destination point. Its advantage is that it can be apply for any possible set of categories in a query. Our work is inspired by the idea of a dynamic programming formulation from Rice et al.'s work, and we extend their algorithm in two aspects. First, we allow multiple categories involved in a computation steps. Second, we introduce "jumping" computations in the DP table that compute an arbitrary row from an other arbitrary row in the table, allowing us freely describe computation steps guided by a directed acyclic graph representation.

Being close to our SRCP query is the optimal route queries with arbitrary order constraints proposed by Li et al. [4]. This query considers partial order constraints over categories, which are 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 [3]. 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).

VII. Conclusions and Future Work

In this paper, we have introduced the general SRCP query for finding the optimal path constrained by categories. We have found a reasonable subset of the general SRCP queries that uses simplified regular expressions as constraints. Even though this subset is limited, it covers all of the existing category-constrained shortest path queries and has efficient implementation. We have proposed a dynamic programming formulation to solve the subset of SRCP queries in which it is reduced to a sequence of single source shortest path searches on the input graph. 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 a directed acyclic graph 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 on road networks, which is called *parallel hardware-accelerated shortest path trees* (PHASE) algorithm [12]. Experiments on complex networks, such as social networks, web graph, biological networks, are also interesting. Another future work is extending the query language so that it can express more constraints on nodes of the optimal path. For example, the current query language cannot describe problems of finding the optimal path that continuously passes multiple *different* nodes in the *same category*. This is a kind of problems to find simple paths (visiting a node at most once); it is interesting but very difficult to solve [6], [16].

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] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *Proceedings of the 9th International Conference on Advances in Spatial and Temporal Databases*, ser. SSTD'05, 2005, pp. 273–290.

[2] M. N. Rice and V. J. Tsotras, "Exact graph search algorithms for generalized traveling salesman path problems," in *Proceedings of the 11th International Conference on Experimental Algorithms*, ser. SEA'12, 2012, pp. 344–355.

[3] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi, "The optimal sequenced route query," *The VLDB Journal*, vol. 17, no. 4, pp. 765–787, Jul. 2008.

[4] J. Li, Y. Yang, and N. Mamoulis, "Optimal route queries with arbitrary order constraints," *IEEE Trans. on Knowl. and Data Eng.*, vol. 25, no. 5, pp. 1097–1110, May 2013.

[5] M. N. Rice and V. J. Tsotras, "Engineering generalized shortest path queries," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13, 2013, pp. 949–960.

[6] C. Barrett, R. Jacob, and M. Marathe, "Formal-language-constrained path problems," *SIAM J. Comput.*, vol. 30, no. 3, pp. 809–837, May 2000.

[7] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proceedings of the 7th International Conference on Experimental Algorithms*, ser. WEA'08, 2008, pp. 319–333.

[8] J. Maue, P. Sanders, and D. Matijevic, "Goal-directed shortest-path queries using precomputed cluster distances," *J. Exp. Algorithmics*, vol. 14, pp. 2:3.2–2:3.27, Jan. 2010.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[11] U. Meyer and P. Sanders, "Delta-stepping: A parallel single source shortest path algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms*, ser. ESA '98, 1998, pp. 393–404.

[12] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "Phast: Hardware-accelerated shortest path trees," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11, 2011, pp. 921–931.

[13] N. Ström, "Automatic continuous speech recognition with rapid speaker adaptation for human/machine interaction," Ph.D. dissertation, KTH, Stockholm, 1997.

[14] H. Bast, D. Delling, A. V. Goldberg, M. Muller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," Tech. Rep. MSR-TR-2014-4, 2014.

[15] M. Sharifzadeh and C. Shahabi, "Processing optimal sequenced route queries using voronoi diagrams," *Geoinformatica*, vol. 12, no. 4, pp. 411–433, Dec. 2008.

[16] A. O. Mendelzon and P. T. Wood, "Finding regular simple paths in graph databases," *SIAM J. Comput.*, vol. 24, no. 6, pp. 1235–1258, Dec. 1995.