Write it Recursively: A Generic Framework for Optimal Path Queries

Akimasa Morihata Kiminori Matsuzaki Masato Takeichi

Graduate School of Information Science and Technology, University of Tokyo, Japan
{morihata,kmatsu}@ipl.t.u-tokyo.ac.jp, takeichi@mist.i.u-tokyo.ac.jp

Abstract

Optimal path queries are queries to obtain an optimal path specified by a given criterion of optimality. There have been many studies to give efficient algorithms for classes of optimal path problem. In this paper, we propose a generic framework for optimal path queries. We offer a domain-specific language to describe optimal path queries, together with an algorithm to find an optimal path specified in our language. One of the most distinct features of our framework is the use of recursive functions to specify queries. Recursive functions reinforce expressiveness of our language so that we can describe many problems including known ones; thus, we need not learn existing results. Moreover, we can derive an efficient querying algorithm from the description of a query written in recursive functions. Our algorithm is a generalization of existing algorithms, and answers a query in $\mathrm{O}(n \log n)$ time on a graph of $\mathrm{O}(n)$ size. We also explain our implementation of an optimal path querying system, and report some experimental results.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program transformation; I.2.8 [*Problem Solving, Control Methods, and Search*]: Dynamic programming; D.3.2 [*Language Classifications*]: Specialized application languages

General Terms Algorithms, Languages

Keywords Optimal path query, Recursive function, Finite state automaton, Program transformation

1. Optimal Path Query

Imagine we are planning a trip to a historic city, in which we intend to look round famous sights. How can we find the best route for strolling the city? We may be able to find the shortest route, because well-known algorithms for shortest path problems will be applicable. However, what we truly want may not be the shortest route in practice: we might want to find a route whose transportation expense is less than some limit; we might want to take a rest in some cafeteria on afternoon; we might want to walk for a while to enjoy the scenery; we might not want to walk after visiting some temple on a hill, etc. After all, it is a complicated problem — how can we find the best route?

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

Our objective is to give a generic framework for *optimal path queries*: we intend to find the optimal route, which is identified by a given criterion of optimality. Optimal path queries are important from both theoretical and practical aspects. In theory, a lot of optimization problems result in routing problems, such as shortest path problems and their variants. In practice, optimal path queries have many real applications. The trip-planning problem described above is a direct application. To give a routing with quality-of-service guarantees is another application (Korkmaz and Krunz 2001), where optimal path queries are necessary to find a routing that satisfies additional requirements such as band width and latency. Querying graph-structured databases is also an application of optimal path queries. Regular path queries (Mendelzon and Wood 1995; Flesca et al. 2006) are known to be an expressive and efficient framework; however, infinite number of paths may match a query, and it is important to extract the optimal one.

Since optimal path queries are important, there are a lot of studies on this topic from algorithmic viewpoints (Joksch 1966; Desrochers and Soumis 1988; Romeuf 1988; Punnen 1991; Barrett et al. 2000; Brodal and Jacob 2004; Villeneuve and Desaulniers 2005; Sherali et al. 2006; Chan and Zhang 2007). However, it is difficult for non-specialists to utilize such studies for their objective. Recall the trip-planning problem. Even finding the shortest route via given sights is a nontrivial problem, and in fact, it is an instance of regular-language-constrained shortest path problems (Romeuf 1988). The first additional requirement, the expense must be less than a limit, is an instance of weight-constrained shortest path problems (Joksch 1966). The second one, take a rest at the cafeteria on afternoon, is a combination of time-window constraints (Desrochers and Soumis 1988) and regular-language constraints. It is not easy to find an existing algorithm that copes with a requirement; furthermore, we need to deal with combinations of many requirements.

In this paper, we propose a generic framework for optimal path queries. We propose a domain-specific language to describe optimal path queries, together with an algorithm to find an optimal path specified in our language. Since we can naturally express combinations of requirements in our language, we are not in trouble to deal with them.

One of the most distinct features of our framework is the use of recursive functions to specify queries. Recursive functions reinforce expressiveness of our language so that we can describe many problems including known ones; thus, we need not learn existing results. Moreover, we can derive an efficient querying algorithm from the description of a query written in recursive functions.

Our contributions are summarized as follows.

A domain-specific language for optimal path queries We design a domain-specific language to describe optimal path queries. In our language, each query is specified by recursive functions on a path.

$$
\begin{array}{rcll}
prog & ::= & \mathsf{minimize}\ f'(x)\ \mathsf{subject\ to}\ p(x)\ \mathsf{where}\ decl \cdots decl & \{\ \text{Program}\ \} \\
decl & ::= & p(\epsilon) = b;\ p(x \cdot a) = \phi; & \{\ \text{Boolean-valued function}\ \} \\
 & | & f(\epsilon) = n;\ f(x \cdot a) = e; & \{\ \text{Integer-valued function}\ \} \\
 & | & h(\epsilon) = n;\ h(x \cdot a) = e'; & \{\ \text{Objective function}\ \} \\
\phi & ::= & b \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid p(x) \mid q(a) \mid e \leq n & \{\ \text{Boolean-valued expression}\ \} \\
e & ::= & n \mid e \oplus e \mid f(x) \mid h(x) \mid w(a) \mid \mathsf{if}\ \phi\ \mathsf{then}\ e\ \mathsf{else}\ e & \{\ \text{Integer-valued expression}\ \} \\
e' & ::= & n \mid e' \oplus e' \mid h(x) \mid w(a) \mid \mathsf{if}\ \phi\ \mathsf{then}\ e'\ \mathsf{else}\ e' & \{\ \text{Expression for the objective function}\ \} \\
\oplus & ::= & + \mid \times \mid \uparrow \mid \downarrow & \{\ \text{Binary operator}\ \}
\end{array}
$$

Figure 1. Syntax of our query language, where $b \in \{True, False\}$ and $n \in \mathbb{Z}_+$ are constant values, $\uparrow$ and $\downarrow$ respectively denote infix binary maximum and minimum operators, $q : E \to \{True, False\}$ is an atomic predicate, $w : E \to \mathbb{Z}_+$ is a weight function, and a and x are respectively variables of type E and E^*.

By virtue of recursive functions, our language is so expressive that it can describe many known problems and their combinations.

A generic and efficient optimal path querying algorithm We derive a generic optimal path querying algorithm from the criterion of optimality described in our language. Our derivation is based on two ideas, *product construction* and *by-need evaluation*, and recursive functions play important roles. In addition, we propose two improvements to remove overheads of the algorithm. Our algorithm is a generalization of some existing algorithms, and finds an optimal path in $\mathrm{O}(n \log n)$ time on a graph of $\mathrm{O}(n)$ size.

A neat optimal path querying system We show an implementation of an optimal path querying system. In our system, querying consists of two stages. The first stage is code generation, where the system analyses the query description and generates codes that consist of information for efficient querying. The second is graph searching, where the system performs optimal path querying using the generated codes. While our implementation is lightweight and consists of only several hundred lines of codes, it can answer queries on graphs of practical size.

The rest of this paper is organized as follows. In Section 2, we prepare basic notations. In Section 3, we describe our query language. In Section 4, we show a derivation of an optimal path querying algorithm. In Section 5, we report our implementation and experiments. We discuss related works in Section 6, and conclude this paper in Section 7. Proofs of lemmas are shown in Appendix A.

2. Preliminary

Given a set A, A^* denotes a set of all sequences whose elements are elements of A. For a sequence $x = [a_0, a_1, \ldots, a_n] \in A^*$ and an element $a \in A$, $x \cdot a$ denotes the extension of x by a, namely $x \cdot a \stackrel{\mathsf{def}}{=} [a_0, a_1, \ldots, a_n, a]$. The empty sequence is denoted by ϵ, and $\epsilon \cdot a$ stands for a singleton sequence $[a]$. A concatenation of two sequences is denoted by $+\!\!+$.

A graph $G = (V, E)$ consists of a set of vertexes V and a set of edges E. Functions $hd : E \to V$ and $tl : E \to V$ respectively return the startpoint and endpoint of an edge. Each edge has weights given by weight functions. Note that there may be more than one weight functions. A sequence of edges $[a_0, a_1, \ldots, a_n] \in E^*$ is said to be a path if $tl(a_k) = hd(a_{k+1})$ holds for all $0 \leq k < n$. A function dst returns the destination of a path, that is, $dst(\epsilon) \stackrel{\mathsf{def}}{=} \bullet$ and $dst(x \cdot a) \stackrel{\mathsf{def}}{=} tl(a)$, where $\bullet$ is a distinguishable value.

We will make use of *deterministic finite state automata* (DFA). A DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, Q_F)$ consists of a finite set of states Q, a finite alphabet Σ, a transition function $\delta : Q \times \Sigma \to Q$, an initial state $q_0 \in Q$, and a set of final states $Q_F \subseteq Q$. For a DFA $\mathcal{A}$ and a state $q' \in Q$, $L_{\mathcal{A}}$ denotes the language accepted by $\mathcal{A}$, and $\mathcal{A}(q')$ denotes the DFA $(Q, \Sigma, \delta, q', Q_F)$.

3. A Language for Optimal Path Queries

3.1 Rules for Query Descriptions

Figure 1 shows the syntax of our query language. At the top of a program, we specify the objective function and the requirement for feasible paths. Both objective functions and requirements are expressed by using recursive functions on a path. There are two kinds of recursive functions: boolean-valued and non-negative-integer-valued functions. Declarations of boolean-valued functions consist of constants, basic boolean operations, function calls, atomic predicates, and inequalities. Right-hand-side expressions of inequalities must be constant non-negative integers. Declarations of integer-valued functions consist of constants, additions, multiplications, minimum and maximum operations, function calls, weights of edges, and conditional expressions. The objective function is an integer-valued function that must not call other functions outside operands of inequalities. Atomic predicates and weights are predefined, and distinguished from recursive functions by types of their arguments: edges or paths.

The semantics the following. Given an objective function h, a requirement p, and a graph G, the program results in one of the minimum-h-valued paths in G among those satisfying p.

In the declaration of each recursive function, we assume that the size of recursive parameters must decrease, such as $f(x \cdot a) = \cdots g(x) \cdots h(x) \cdots$; hence all recursive functions trivially terminate. Declarations like $f(x \cdot a) = \cdots g(x \cdot a) \cdots$ are prohibited, and we need to unfold $g(x \cdot a)$. Meanwhile, we use such declarations as a syntactic sugar. Declarations like $f(x \cdot a \cdot b) = \cdots g(x) \cdots$ are also prohibited, and we need to use an auxiliary function: $f(x' \cdot b) = \cdots f'(x') \cdots ; \cdots ; f'(x \cdot a) = \cdots g(x) \cdots$. In addition, we impose the following assumption, which is required to utilize Dijkstra algorithm.

ASSUMPTION 3.1. *For an objective function h and any path $x \cdot a$, $h(x) \leq h(x \cdot a)$ holds.* □

It is not difficult to confirm Assumption 3.1 from descriptions of objective functions in a sound manner. Given the objective function h, the following function $check$ takes the declaration of its recursive case and returns $True$ only if it satisfies Assumption 3.1.

$$
\begin{array}{lcl}
check[\![h(x \cdot a) = e';]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e']\!] \\
chk_h[\![n]\!] & \stackrel{\mathsf{def}}{=} & False \\
chk_h[\![e'_1 + e'_2]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e'_1]\!] \vee chk_h[\![e'_2]\!] \\
chk_h[\![e'_1 \times e'_2]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e'_1]\!] \wedge chk_h[\![e'_2]\!] \\
chk_h[\![e'_1 \uparrow e'_2]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e'_1]\!] \vee chk_h[\![e'_2]\!] \\
chk_h[\![e'_1 \downarrow e'_2]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e'_1]\!] \wedge chk_h[\![e'_2]\!] \\
chk_h[\![h(x)]\!] & \stackrel{\mathsf{def}}{=} & True \\
chk_h[\![w(a)]\!] & \stackrel{\mathsf{def}}{=} & False \\
chk_h[\![\mathsf{if}\ \phi\ \mathsf{then}\ e'_1\ \mathsf{else}\ e'_2]\!] & \stackrel{\mathsf{def}}{=} & chk_h[\![e'_1]\!] \wedge chk_h[\![e'_2]\!]
\end{array}
$$

Recall that all weights and constant integers are non-negative; therefore, for the case of addition, it is sufficient to check that an operand contains at least one recursive call. This condition is not sufficient for the case of multiplication because of the possibility of a multiplication with 0. Therefore, we check that both operands contain a recursive call, because $y \geq x \wedge z \geq x$ implies $y \times z \geq x$ for any $x \in \mathbb{Z}_+$. The other cases are trivial.

3.2 Running Examples

Shortest Path Problems with Transit Costs

Let us consider a shortest path problem with transit costs, in which we need to pay an additional cost to ride on a train, such as waiting time. The following is a description of the problem in our language, where the starting point is s, the destination is t, the additional cost is C.

$$
\begin{array}{l}
\mathsf{minimize}\ cost(x)\ \mathsf{subject\ to}\ p(x)\ \mathsf{where} \\
\quad p(\epsilon) \ = \ False; \\
\quad p(x \cdot a) \ = \ start_s(x \cdot a) \wedge to_t(a); \\
\quad start_s(\epsilon) \ = \ False; \\
\quad start_s(x \cdot a) \ = \ start_s(x) \vee (empty(x) \wedge from_s(a)); \\
\quad empty(\epsilon) \ = \ True; \\
\quad empty(x \cdot a) \ = \ False; \\
\quad walk(\epsilon) \ = \ True; \\
\quad walk(x \cdot a) \ = \ \neg train(a); \\
\quad cost(\epsilon) \ = \ 0; \\
\quad cost(x \cdot a) \ = \ cost(x) + w(a) \\
\qquad\qquad + (\mathsf{if}\ walk(x) \wedge train(a)\ \mathsf{then}\ C\ \mathsf{else}\ 0)
\end{array}
$$

In the description, w is a weight function. The functions $from_s$, to_t, and $train$ are atomic predicates. The definitions of $from_s$ and to_t are $from_s(e) \stackrel{\mathsf{def}}{=} (hd(e) = s)$ and $to_t(e) \stackrel{\mathsf{def}}{=} (tl(e) = t)$, respectively. The predicate $train$ checks whether we are riding on a train.

Use of our language is not restricted to point-to-point optimal path problems, and we need to examine two endpoints of paths explicitly. The recursive function $start_s$ checks whether a path starts from s by using an auxiliary function $empty$.

The objective function $cost$, which is the characteristic part of this problem, uses a recursive function $walk$ to check the necessity of transit costs. The function $walk$ checks whether we have not been riding on a train.

Length-Constrained Shortest Path Problems

Next is a length-constrained shortest path problem, in which the objective is to find the shortest path that consists of fewer edges than a given limit. The following is a description of a length-constrained shortest path problem, where the starting point is s, the destination is t, and the limit is K.

$$
\begin{array}{l}
\mathsf{minimize}\ wsum(x)\ \mathsf{subject\ to}\ p(x)\ \mathsf{where} \\
\quad p(\epsilon) \ = \ False; \\
\quad p(x \cdot a) \ = \ start_s(x \cdot a) \wedge to_t(a) \wedge (len(x \cdot a) \leq K); \\
\quad start_s(\epsilon) \ = \ False; \\
\quad start_s(x \cdot a) \ = \ start_s(x) \vee (empty(x) \wedge from_s(a)); \\
\quad empty(\epsilon) \ = \ True; \\
\quad empty(x \cdot a) \ = \ False; \\
\quad wsum(\epsilon) \ = \ 0; \\
\quad wsum(x \cdot a) \ = \ wsum(x) + w(a); \\
\quad len(\epsilon) \ = \ 0; \\
\quad len(x \cdot a) \ = \ len(x) + 1;
\end{array}
$$

The recursive functions $start_s$ and $empty$ are the same as the previous example. The integer-valued function len computes the number of edges used. The objective function $wsum$ computes the summation of weights of edges.

3.3 Expressiveness

While our language is simple, it can express a large number of known problems. Here we enumerate some of them. It is worth noting that their combinations can be expressed in our language. We assume only non-negative integers are used in descriptions. For a large number of problems, negative weights of edges can be removed in safe by re-weighting method proposed by Johnson (1977).

FACT 3.2. The following problems can be expressed in our language: point-to-point shortest path problems; weight-constrained shortest path problems (Joksch 1966) (find the minimum-cost path whose weight is less than a given limit); shortest path problems with time windows (Desrochers and Soumis 1988) (find the shortest path where each vertex can be used only during its time window); regular-language-constrained shortest path problems (Romeuf 1988) (find the shortest path whose label is in a given regular language); time-table queries (Brodal and Jacob 2004) (regular-language-constrained shortest path problems where weights of edges depend on time); shortest path problems with forbidden paths (Villeneuve and Desaulniers 2005) (find the shortest path that does not contain given forbidden paths); approach-dependent, time-dependent, label-constrained shortest path problems (Sherali et al. 2006) (time-table queries where weights of edges depend on their preceding vertexes). □

Since recursive functions are available, it is not difficult to describe these problems in our language. For example, regular-language-constrained shortest path problems can be expressed in our language based on the binary encoding technique: arrange a set of boolean-valued functions to simulate the transition function of the DFA corresponding to the given regular language.

While our language is expressive, there are several problems that cannot be dealt with. For example, we cannot describe maximization problems such as maximum capacity path problems (Punnen 1991). Context-free-language-constrained shortest path problems (Barrett et al. 2000) (find the shortest path whose label is in a given context-free language) are also out of the range, because recursive functions must traverse over a path in the fixed direction in our language.

4. Deriving an Optimal Path Querying Algorithm

In this section, we show an optimal path querying algorithm with two improvements. We outline our method in Section 4.1, and give detailed explanations in Sections 4.2 and 4.3: we reduce optimal path problems into shortest path problems in Section 4.2, and introduce our optimal path querying algorithm in Section 4.3. In addition to the algorithm, we propose two improvements in Section 4.4. We compare our algorithm with others in Section 4.5, and show some examples in Section 4.6.

Here we prepare some notations. For integer-valued functions f and g in a query description, $f \rightsquigarrow g$ denotes that the definition of g syntactically contains function calls of f in its non-predicate part. For example, $f \rightsquigarrow g$ holds when $g(x \cdot a) = \mathsf{if}\ \cdots\ \mathsf{then}\ \cdots f(x) \cdots\ \mathsf{else}\ \cdots$, and $f \rightsquigarrow g$ may not hold when $g(x \cdot a) = \mathsf{if}\ (f(x) \leq n)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots$. The transitive closure of $\rightsquigarrow$ is denoted by $\stackrel{+}{\rightsquigarrow}$. Intuitively $f \stackrel{+}{\rightsquigarrow} g$ means a result of g would contain a result of f. We use the same notation for integer-valued expressions by regarding them as integer-valued functions.

4.1 Basic Idea

First, we sketch our derivation of an optimal path querying algorithm. We derive the algorithm by reducing an optimal path problem into a usual shortest path problem. The key is the notion of case analyses: we first find out case analyses that are necessary for deter-

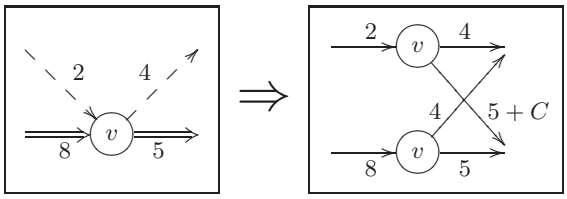

Figure 2. Reducing the shortest path problem with transit costs into a usual shortest path problem. Double-lines arrows stand for edges of riding on a train, broken arrows stand for edges of walking, and C stands for the cost for a transit.

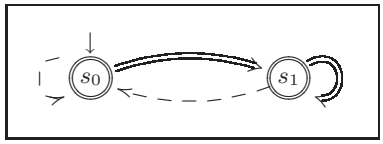

Figure 3. A DFA representing the structure of case analyses of the shortest path problem with transit costs. The state s_1 corresponds to the situation of riding on a train, and s_0 corresponds to the others. Double-lines arrows are transitions when the DFA takes an edge of riding on a train, and broken arrows are transitions when it takes an edge of not riding on a train. s_0 is the initial state, and both s_0 and s_1 are final states.

mining optimality of paths, and then, we construct a graph whose shortest path is the optimal path of the original graph.

Consider the shortest path problem with transit costs, which we have seen in Section 3.2. Notice that for determining optimality, it is necessary to distinguish whether we are riding on a train or not, because it affects the value of the objective function. Based on this case analysis, we can reduce the problem into a shortest path problem, as shown in Figure 2. To distinguish these two cases, we divide the node v into two: one corresponds to the case of riding on a train, and the other corresponds to the case of not riding on a train. Then, the shortest path on the right graph is the optimal path on the left graph.

Although this approach is applicable for a large set of problems, there are two issues. The first issue is correctness. How can we give an appropriate reduction? Here we give a systematic method based on *product construction*. We construct a DFA that represents the structure of case analyses, and derive a reduction by making a product of the DFA and the underlying graph. The second issue is efficiency. In general, the reduction makes the underlying graph much larger and derives practically inefficient algorithms. We solve the second issue by *by-need evaluation*, that is, we delay the construction of the product as possible. We will give a detailed explanation of by-need evaluation in Section 4.3, and we focus on product construction in the following.

Now let us demonstrate our method with the example, shortest path problems with transit costs. First, notice that the recursive function $walk$ in the query description retains the information corresponding to the case analysis that is necessary to determine optimality of paths. It is not accidental, and we can always find sufficient information from recursive functions. Because the values of recursive functions for x (e.g. $walk(x)$) are sufficient to compute values for $x \cdot a$ (e.g. $cost(x \cdot a)$), the values for x give sufficient information to determine optimality of the case of $x \cdot a$.

Next we specify the recursive function by a transition function of a DFA. Figure 3 shows the DFA corresponds to the recursive function $walk$. Symbols of the DFA are edges. The initial state is s_0, which corresponds to the case where $walk$-value is $True$, that is, we are not on a train. When we ride on a train, the state becomes s_1, which corresponds to the case where $walk$-value is $False$. It turns into s_0 when we get off a train. Both s_0 and s_1 are final states, because $walk$-values do not affect to feasibility.

Finally, we reduce the problem into a shortest path problem. As mentioned, product construction works well. The product of the DFA in Figure 3 and the underlying graph (e.g., the left of Figure 2) yields the graph that gives a reduction into a shortest path problem (e.g., the right of Figure 2). The weights of edges are given by the objective function $cost$. After all, we can solve the problem.

4.2 Reducing Optimal Path Problems into Shortest Path Problems

Now let us consider generic cases, where let $\{p_0, p_1, \ldots, p_m\}$ and $\{f_0, f_1, \ldots, f_n\}$ be respectively the whole set of boolean-valued functions and integer-valued functions (including the objective function) in the description.

As mentioned, we would like to construct a DFA that represents the structure of case analyses by regarding the set of recursive functions as a transition function of the DFA. However, a recursive function cannot be a transition function of the DFA when the size of its range is infinite, because the range of the recursive function corresponds to the set of states of the DFA. This finiteness guarantees the termination of the algorithm and thus indispensable.

To resolve this problem, we set a cut-off to each integer-valued function. For each integer-valued function f, we define its cut-off version $\hat{f} : E^* \to (\mathbb{Z}_+ \cup \{\infty\})$ as follows, where a set I stands for all inequalities in the description and we regard $\max(\emptyset)$ as $-\infty$.

$$\hat{f}(x) \stackrel{\text{def}}{=} \begin{cases} f(x) & \text{if } f(x) \leq \max\{\, n \mid (e \leq n) \in I \wedge f \stackrel{+}{\rightsquigarrow} e \,\} \\ \infty & \text{otherwise} \end{cases}$$

The value of $\hat{f}$ is the same as that of f up to the boundary and turns into ∞ when it exceeds the boundary. The boundary for $\hat{f}$ is the maximum right-hand-side value of the inequality whose left-hand-side value would include some $\hat{f}$ value. Although the function $\hat{f}$ forgets some information of f, it is no problem: when values that are larger than boundaries are used, integer-valued expressions result in larger values and inequalities yield false; therefore, their exact values are needless.

Now let us construct a DFA. The following auxiliary function $state$ corresponds to the transition function of the DFA.

$$state(x) \stackrel{\text{def}}{=} (\hat{f}_0(x), \ldots, \hat{f}_n(x), p_0(x), \ldots, p_m(x))$$

As required, the range of the function $state$ is finite. Once the function $state$ is given, it is easy to construct the DFA specifying the structure of case analyses: The symbols are edges; The initial state is $state(\epsilon)$; The transition function δ is defined as $\delta(state(x), a) = state(x \cdot a)$; The set of final states S_F is defined by $S_F = \{\, state(x) \mid x \in E^* \wedge p(x) \,\}$, where p is the requirement for feasible paths. Then, the DFA accepts all sequences of edges each of which is feasible if it is a path. It is worth mentioning that the above construction certainly yields a DFA because the δ is a function, as the following lemma shows.

LEMMA 4.1. If $state(x) = state(y)$ holds for two sequences x and y, then $state(x \cdot a) = state(y \cdot a)$ holds for any edge a. □

Furthermore, even though the DFA above forgets some information of recursive functions, it certainly retains sufficient information of case analyses, as the following lemmas show.

LEMMA 4.2. For an edge a and two paths x and y such that $state(x) = state(y)$ and $dst(x) = dst(y)$ hold, $x \cdot a$ is a feasible path if and only if $y \cdot a$ is a feasible path. □

LEMMA 4.3. For the objective function h and two sequences x and y, assume that both $state(x) = state(y)$ and $h(x) \leq h(y)$ hold. Then, for any edge a, $h(x \cdot a) \leq h(y \cdot a)$ holds. □

Lemmas 4.1, 4.2, and 4.3 tell us that it is sufficient to retain the minimum-weighted path among those having the same $state$-value: Given the objective function h, for any paths x and y such that all of $state(x) = state(y)$, $dst(x) = dst(y)$, and $h(x) \leq h(y)$ hold, $y +\!\!+ z$ is a feasible path only if $x +\!\!+ z$ is a feasible path; moreover, $h(x +\!\!+ z) \leq h(y +\!\!+ z)$ holds.

Now that we derived the required DFA, which is expressed by the function $state$, product construction enables us to solve optimal path problems.

4.3 Optimal Path Querying Algorithm

To give our optimal path querying algorithm, we prepare an auxiliary function $pstate$ defined as follows.

$$pstate(x) \stackrel{\mathsf{def}}{=} (\hat{f}_0(x), \ldots, \hat{f}_n(x), p_0(x), \ldots, p_m(x), dst(x))$$

Roughly speaking, $pstate$ corresponds to the transition function of the product of $state$ and the underlying graph. Notice that $pstate(x) = pstate(y)$ is equivalent to $state(x) = state(y) \wedge dst(x) = dst(y)$.

Our optimal path querying algorithm, which corresponds to Dijkstra algorithm on the product, is the following, where h is the objective function.

ALGORITHM $\mathcal{A}$ (Optimal path querying algorithm).
Input: A graph.
Output: An optimal path if it exists.
(1) Let W be $\{\epsilon\}$ and N be $\emptyset$.
(2) Exit if $W = \emptyset$. *// There exists no feasible path.*
(3) Extract the minimum-h-valued path x from W.
(4) If x is feasible, return x. *// x is an optimal path.*
(5) Add $pstate(x)$ to N.
(6) For each path $z \in \{\, x \cdot a \mid a \in E \,\}$,
  (6-a) If $pstate(z) \in N$, do nothing.
  (6-b) If $\forall y \in W : pstate(z) \neq pstate(y)$, add z to W.
  (6-c) If $\exists y \in W : pstate(z) = pstate(y) \wedge h(z) < h(y)$, replace y by z.
(7) Go to (2). □

THEOREM 4.4. Algorithm $\mathcal{A}$ always terminates and returns an optimal path if it exists.

Proof . Notice that W never contains two paths whose $pstate$ values are the same. Moreover, W never contains a path whose $pstate$ value is in N. Therefore, the size of N increases strictly. Since N is always a subset of the range of $pstate$, which is finite, the algorithm terminates.

From Lemmas 4.1, 4.2, and 4.3, it is sufficient to consider extensions of the minimum-h-valued path for each equivalent set raised from the value of $pstate$; thus, paths discarded in the step (6-c) are unnecessary. From Assumption 3.1, paths are examined in increasing order of their h-values in the step (4); thus, paths discarded in the step (6-a) are also unnecessary, because another path of the same $pstate$-value and less (or equal) h-value was considered before. In summary, the algorithm is correct. □

As explained, each problem is reduced into a shortest path problem on a larger graph, which is the product of $state$ and the underlying graph. However, explicit construction of the larger graph is inefficient, especially from the viewpoints of computational space. Algorithm $\mathcal{A}$ generates a vertex of the product when the algorithm runs across it, and terminates when an optimal path is found. Therefore, Algorithm $\mathcal{A}$ constructs the product only if it is necessary to find the optimal path. It is worth noticing that the recursive function $pstate$ gives a compressed representation of the product. Hence, we can extract information of the larger graph in a by-need manner and accomplish querying efficiently.

Data Structures for Efficient Implementation

While an ideal hash set gives an efficient implementation of N, the priority queue W requires a bit complicated data structure. The data structure needs to support efficient implementation of inserting an element, extracting the minimum-weighted element, decreasing the weight of an element, and finding an element of the specified $pstate$-value. We prepare a Fibonacci heap with an ideal hash map for W. The Fibonacci heap stores paths according to their weights. The hash map associates each $pstate$-value to the element having the value in the Fibonacci heap. Notice that we should rearrange pointers in the hash set after the Fibonacci heap is manipulated. Therefore, we prepare back pointers from elements in the Fibonacci heap to the entries of the hash map, and keep their consistency.

After all, operations for W except extract-min are (amortized) constant time. Extracting the minimum-weighted element from W takes $\mathrm{O}(\log n)$ time, where n is the size of W.

Computational Complexity

Let k be the size of the range of $state$; then the size of the range of $pstate$ is at most $k|V|$. We assume that all results of recursive functions are memoized.

In Algorithm $\mathcal{A}$, the steps (2) to (7) are executed at most $k|V|$ times. Each execution of the steps (2) to (5) costs $\mathrm{O}(\log(k|V|))$ time. Each execution of the step (6) costs amortized $\mathrm{O}(1)$ time for a path (z). The number of such paths is at most $k|E|$, because each edge is used at most k times. In summary, the time complexity of Algorithm $\mathcal{A}$ is $\mathrm{O}(k|V| \log(k|V|) + k|E|)$.

It is worth noticing that the value k depends only on the description of the query; hence the time complexity of Algorithm $\mathcal{A}$ is polynomial in the size of the graph. However, the value k would be exponential to the size of the description. For example, a traveling salesman problem requires a description of at least $\mathrm{O}(|V|)$ size, and then the value k becomes $\mathrm{O}(2^{|V|})$; thus, it is an exponential time algorithm, yet it is much more efficient than the trivial algorithm that takes $\mathrm{O}(|V|!)$ time.

4.4 Improving Efficiency of the Optimal Path Querying Algorithm

We have shown an algorithm to find the optimal path. While the algorithm is correct, there is room for improvement. For example, when we want to find the shortest path from a vertex s, paths whose startpoints are not s are useless. However, naive execution of Algorithm $\mathcal{A}$ results in enumeration of such useless paths. Here we give two improvements to remove such inefficiency.

The notion of DFA is useful to discuss such improvements, and we give improvements by examining the DFA represented by $state$. It is worth noticing that minimization of the DFA represented by $state$ is incorrect. For two paths x and y, assume that $x +\!\!+ z$ is feasible if and only if $y +\!\!+ z$ is feasible. Then, we may attempt to merge $state(x)$ with $state(y)$. However, $h(x) \leq h(y)$ may not imply $h(x +\!\!+ z) \leq h(y +\!\!+ z)$, where h is the objective function, because different conditional branches may be used to compute $h(x +\!\!+ z)$ and $h(y +\!\!+ z)$; thus, merging $state(x)$ with $state(y)$ will break the property in Lemma 4.3. In other words, we need to be careful about the branches in the objective function.

To simplify the discussion, we assume the objective function h is declared using guarded expressions, instead of conditional expressions, as follows.

$$\begin{array}{lcl} h(\epsilon) & = & n; \\ h(x \cdot a) & = & (\{\phi_1\} \Rightarrow e'_1)\,(\{\phi_2\} \Rightarrow e'_2) \cdots (\{\phi_m\} \Rightarrow e'_m); \end{array}$$

Each e'_i must not include any conditional expressions. The above equation means that the value of $h(x)$ is that of $e'_i(x)$ when $\phi_i(x)$ holds. It is easy to translate conditional expressions into guarded expressions.

Now let $\mathcal{S} = (S, E, \delta, s_0, S_F)$ be the DFA that corresponds to $state$. We label each edges by an integer $i \in \{1, \ldots, m\}$, which stands for the branch used to compute the objective function. Then, $\mathcal{S}' = (S, E \times \{1, \ldots, m\}, \delta', s_0, S_F)$ also forms a DFA, where the transition function δ' is defined as follows.

$$\delta'(state(x), (a, i)) \stackrel{\text{def}}{=} state(x \cdot a) \text{ if } \phi_i(x \cdot a)$$

Our improvements are expressed by the following two lemmas.

LEMMA 4.5. If $L_{\mathcal{S}'(state(x))} = \emptyset$ holds for a sequence x, $x +\!\!+ y$ is not feasible for any sequence y. □

LEMMA 4.6. For the objective function h and two sequences x and y, assume all of $L_{\mathcal{S}'(state(x))} \supseteq L_{\mathcal{S}'(state(y))}$, $dst(x) = dst(y)$, and $h(x) \leq h(y)$ hold; then, for any sequences z, $x +\!\!+ z$ is feasible if $y +\!\!+ z$ is feasible, and $h(x +\!\!+ z) \leq h(y +\!\!+ z)$ holds. □

Lemmas 4.5 and 4.6 enable us to find unnecessary paths: they respectively show how to find paths whose extensions yield no feasible path and those whose extensions yield no better path than others. We can implement these improvements by implementing checks on emptiness and inclusion of regular languages.

Notice that Lemma 4.6 includes minimization of $\mathcal{S}'$. However, naive implementation of Lemma 4.6 is inefficient, because it is not easy to find better (or worse) paths from the priority queue; hence, minimization of $\mathcal{S}'$ would be appropriate in general. Lemma 4.6 is useful for more restrictive settings where it is easy to point out better/worse paths than the given path.

It is worth noting that we can achieve improvements without information of the input graph by considering constraints on edges instead of edges: for example, instead of an edge a from a vertex v_1 to a vertex v_2 of weight n, it is sufficient to consider a constraint $\lambda a.\, hd(a) = v_1 \wedge tl(a) = v_2 \wedge w(a) = n$.

4.5 Correspondence to Existing Algorithms

Dijkstra-like algorithms have been proposed for classes of optimal path problems, and some of them are equivalent to Algorithm $\mathcal{A}$ with the improvements, when the specification of the problem can be written in our language. In other words, our algorithm is a generalization of existing algorithms.

FACT 4.7. Algorithm $\mathcal{A}$ with the improvements of Lemmas 4.5 and 4.6 is equivalent to the following algorithms except for implementation of the priority queue: Dijkstra algorithm for point-to-point shortest path problems; the generalized permanent labeling algorithm of Desrochers and Soumis (1988) for shortest path problems with time windows; the heap-Dijkstra algorithm of Sherali et al. (2006) for approach-dependent, time-dependent, label-constrained shortest path problems. □

In the algorithms above, like ours, problems are reduced into shortest path problems and solved by implicit application of Dijkstra algorithm. Our construction of $pstate$ certainly corresponds to their reductions, and overheads are removed by our improvements.

4.6 Examples

Shortest Path Problem with Transit Costs

From the description of the shortest path problem with transit costs, seen in Section 3.2, the function $state$ is obtained as follows.

$$state(x) \stackrel{\text{def}}{=} (\widehat{cost}(x), p(x), start_s(x), empty(x), walk(x))$$
$$\widehat{cost}(x) \stackrel{\text{def}}{=} \infty$$

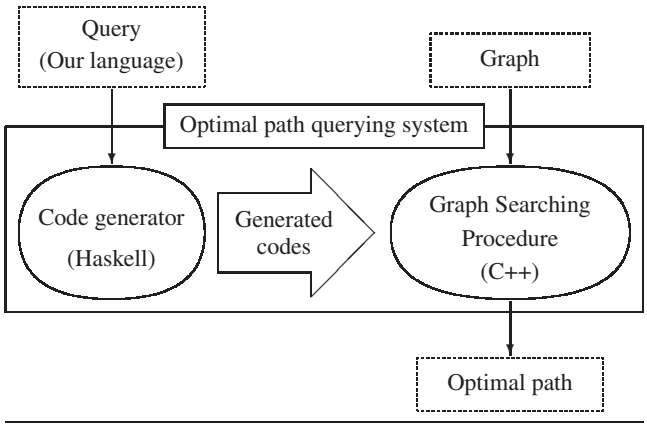

Figure 4. Overview of our system

The values of $\widehat{cost}$, p, $empty$ are not essential: $\widehat{cost}$ always returns ∞ because of absence of inequalities; p always yields false until an optimal path is found; $empty$ always yields false for any non-empty paths. In summary, the auxiliary function $state$ distinguishes paths by values of $start_s$ and $walk$.

Lemma 4.5 tells us that a path x is unnecessary if neither $empty(x)$ nor $start_s(x)$ holds, which corresponds to the case where x does not start from the vertex s. Thus, the algorithm enumerates paths starting from s, and paths are distinguished based on the current vertex and whether we are riding on a train. The number of paths considered in the step (6) of Algorithm $\mathcal{A}$ is at most $4|V| + 2$, which becomes $2|V| + 2$ after the improvements.

Length-Constrained Shortest Path Problem

The $state$ for a length-constrained shortest path problem is obtained as follows.

$$state(x) \stackrel{\text{def}}{=} (\widehat{wsum}(x), \widehat{len}(x), p(x), start_s(x), empty(x))$$
$$\widehat{wsum}(x) \stackrel{\text{def}}{=} \infty$$
$$\widehat{len}(x) \stackrel{\text{def}}{=} \begin{cases} len(x) & \text{if } len(x) \leq K \\ \infty & \text{otherwise} \end{cases}$$

As the same as the previous example, values of $\widehat{wsum}$, p, and $empty$ are not important. Paths are distinguished by the values of $start_s$ and $\widehat{len}$: whether the path starts from s and how much edges are used (or, whether more than K edges are used).

Lemma 4.5 tells us that a path x is unnecessary if neither $empty(x)$ nor $start_s(x)$ holds or $\widehat{len}(x)$ is ∞. The former is the case where x does not start from s, and the latter is the case that x consists of more than K edges. Lemma 4.6 tells us that a path x is unnecessary if there exists a path y such that all of the following four hold: the destinations of x and y are the same; all values of p, $start_s$, $empty$ are the same for x and y; $\widehat{len}(x) \geq \widehat{len}(y)$; $wsum(x) \geq wsum(y)$. This is the case that the $wsum$-value of x is worse than that of y while x uses more edges than y.

The number of paths considered in the step (6) of Algorithm $\mathcal{A}$ is at most $2(K + 2)|V| + 2$, which becomes $(K + 1)|V| + 2$ after the improvements.

5. Optimal Path Querying System

In this section, we report our optimal path querying system and experimental results. The system is available from the first author's website[1].

[1] `http://www.ipl.t.u-tokyo.ac.jp/~morihata/OPQ.tar.gz`

```
minimize wsum(x)
s.t. len(x) <= 20 && st0(x) && to1(x)
  where
  wsum() = 0;
  wsum(x.e) = w(e) + wsum(x);
  len() = 0;
  len(x.e) = 1 + len(x);
  empty() = true;
  empty(x.e) = false;
  st0() = false;
  st0(x.e) = st0(x) || (empty(x) && from(e,0));
  to1() = false;
  to1(x.e) = to(e,1);
```

Figure 5. A query description of a length-constrained shortest path problem.

5.1 Implementation

Figure 4 shows the overview of our system. Querying on our system consists of two stages. The first stage is code generation, where the system analyses the query description and generates codes that consist of information for efficient querying. The second is graph searching, where the system performs Algorithm $\mathcal{A}$ using the generated codes.

This two-staged implementation improves modularity. First, these two stages are essentially independent: the first stage only cares about descriptions of queries, and the second one concentrates on searching on graphs. Moreover, since Algorithm $\mathcal{A}$ is essentially Dijkstra algorithm, though a bit generalized, it is possible to substitute another library for the second stage. Separating these two would be practical to use our system as a component of a system.

Code Generator

The code generator is made of three hundred lines of Haskell codes excluding codes for the parser. It generates C++ codes, which mainly includes the following four: the definitions of recursive functions, the definition of the auxiliary function $pstate$, the requirement for feasible paths, and the improvements.

We only implemented the improvement of Lemma 4.5. To make the computation of this part faster, we binarize each integer-valued function by whether the value exceeds the boundary, and decreases the size of the automaton.

Graph Searching Procedure

Algorithm $\mathcal{A}$ is implemented in C++. As mentioned, this part may be replaced by another library. The program is made of three hundred lines of codes.

In the implementation, we use a heap instead of a Fibonacci heap, because Fibonacci heaps are inefficient in practice. Hence the time complexity is $\mathrm{O}(k|E|\log(k|V|))$, where k is the size of the range of $state$.

5.2 Example

Figures 5 and 6 respectively show the query description and the generated C++ codes for a length-constrained shortest path problem. In the query description in Figure 5, both `from(e,0)` and `to(e,1)` are atomic predicates, where 0 and 1 are identifiers of vertexes. In the codes in Figure 6, a constructor `val(val v,edge e)` performs the computations and memoizations of recursive functions. The definition of $pstate$ is encoded as a function object `val_eq_t` and a hash function `val_hash_t`. The former is for heaps, and the latter is for hash sets. The function `constraint` is the function to check feasibility. The function `unnecessary` is the function to find unnecessary paths, which is obtained from the improvement of Lemma 4.5.

```
#include "node_edge.h"

#define PRIME_FOR_HASH 402653189
#define PRIME_FOR_HASH_ 1610612741

struct val {
 int wsum;
 int len;
 bool empty;
 bool st0;
 bool to1;
 node n;
 val() {
  n = -1;
  wsum = 0;
  len = 0;
  empty = true;
  st0 = false;
  to1 = false;
 }
 val(val v,edge e) {
  wsum = (e.w+v.wsum);
  len = (1+v.len);
  empty = false;
  st0 = (v.st0 || (v.empty && e.in==0));
  to1 = e.out==1;
  n = e.out;
 }
 int weight() const { return wsum; }
 static val ninf() {
  val v;
  v.wsum = INT_MIN;
  return v;
 }
};

struct val_hash_t {
 unsigned long operator()(const val &v) const {
  unsigned long long k = v.n;
  k = (k * PRIME_FOR_HASH + v.empty) % PRIME_FOR_HASH_;
  k = (k * PRIME_FOR_HASH + v.st0) % PRIME_FOR_HASH_;
  k = (k * PRIME_FOR_HASH + v.to1) % PRIME_FOR_HASH_;
  k = (k * PRIME_FOR_HASH + ((v.len<=20)?v.len:21))
     % PRIME_FOR_HASH_;
  return (unsigned long)k;
 }
} val_hash;

struct val_eq_t {
 bool operator()(const val &v, const val &w) const {
  return (
   v.n == w.n &&
   v.empty == w.empty &&
   v.st0 == w.st0 &&
   v.to1 == w.to1 &&
   ((v.len<=20)?v.len:21) == ((w.len<=20)?w.len:21)
  );
 }
} val_eq;

inline bool constraint( val v ) {
 return ((v.len<=20) && (v.st0 && v.to1));
}

inline bool unnecessary( val a ) {
 return (
  (a.empty && a.st0 && a.to1 && (a.len>20)) ||
  (a.empty && a.st0 && (!a.to1) && (a.len>20)) ||
  (a.empty && (!a.st0) && a.to1 && (a.len>20)) ||
  (a.empty && (!a.st0) && (!a.to1) && (a.len>20)) ||
  ((!a.empty) && a.st0 && a.to1 && (a.len>20)) ||
  ((!a.empty) && a.st0 && (!a.to1) && (a.len>20)) ||
  ((!a.empty) && (!a.st0) && a.to1 && (a.len<=20)) ||
  ((!a.empty) && (!a.st0) && a.to1 && (a.len>20)) ||
  ((!a.empty) && (!a.st0) && (!a.to1) && (a.len<=20)) ||
  ((!a.empty) && (!a.st0) && (!a.to1) && (a.len>20))
 );
}
```

Figure 6. The C++ codes generated from the query in Figure 5.

	RAND1	RAND2	RAND3	RAND4	NY	FLA	CAL	EUSA
$\|V\|$	$131,072$	$131,072$	$1,048,576$	$1,048,576$	$264,346$	$1,070,376$	$1,890,815$	$3,598,623$
$\|E\|$	$524,288$	$2,097,152$	$2,097,152$	$4,194,304$	$733,846$	$2,712,798$	$4,657,742$	$8,778,114$

Table 1. Size of graphs

Query (#state)	RAND1	RAND2	RAND3	RAND4	NY	FLA	CAL	EUSA
SP-boost	0.11	0.24	1.04	1.55	0.10	0.48	1.06	2.60
SP (1)	0.31	0.77	1.68	3.14	0.29	1.29	2.67	6.52
3-SP (2)	0.84	2.24	3.95	9.11	0.88	4.12	8.90	21.82
TRANS (2)	0.38	1.02	2.02	3.94	0.40	1.79	3.80	9.60
TLEN (42)	1.52	3.36	28.42	20.08	8.10	10.06	14.99	25.02

Table 2. Experimental results (unit: second): the bracketed numbers shows the number of essential states of the DFA $state$.

5.3 Experiments

The environment of our experiments is the following: dual Intel Quad-Core Xeon 3.0GHz CPUs; 8GB memory; Mac OS X; g++ 4.2.2 and ghc 6.6.1. Only one core was used while the machine had eight cores.

The following four queries were used: SP (find the point-to-point shortest path); 3-SP (find the point-to-point shortest path that passes another given vertex); TRANS (find the shortest path with transit costs); TLEN (find the shortest path that uses less or equal to twenty edges of riding on a train). For each specification, the code generation step finished immediately (less than 0.1 second). In addition to them, we prepare an implementation of the point-to-point shortest path querying for comparison. The implementation is based on Dijkstra algorithm in C++ Boost Graph Library (Siek et al. 2001), and denoted by "SP-boost".

We used eight graphs. We generated four graphs randomly, where the startpoint, endpoint, and weight of each edge were given uniformly. These four are denoted by RAND1 (relatively small), RAND2 (dense), RAND3 (sparse), and RAND4 (relatively large). We borrowed four graphs from the benchmarks of the 9th DIMACS implementation challenge[2]. The four used were travel time data of NY (New York City), FLA (Florida), CAL (California and Nevada), and EUSA (Eastern USA). The sizes of graphs are shown in Table 1. In these graphs, edges had no category information (such as "train"), and we added it in an ad-hoc manner. We regarded each node whose identifier is odd as a station and each edge from a station to a station as an edge of riding on a train. For each graph, we uniformly generated 1000 pairs of a starting point and a destination (and another vertex for 3-SP) of feasible paths, and measured the average computational times.

The results are shown in Table 2. The bracketed numbers in the first column are the numbers of states of the DFA $state$ obtained after applying the improvement of Lemma 4.5, and show the theoretical ratios of computation times. To see precise ratios, we count the number of "essential" states, that is, we neglect states that represent only the empty path or the optimal path. The other columns show computational times excluding times for inputting the graph.

On one hand, even for the road network of eastern USA, our system returned results of queries in a minute, which is only several times slower than the point-to-point shortest path querying by an existing library. It shows promise of our framework. On the other hand, SP runs two or three times slower than SP-boost. The difference is the overhead of generality. Especially, Algorithm $\mathcal{A}$ requires a data structure that is a bit more complicated than an ordinary heap, as discussed in Section 4.3, and it affects efficiency. It is worth noticing that the ratio does not go worse when the graph becomes larger.

The results indicate that more detailed experiments would be necessary for application-specific uses of our framework. First, real computational times are not exactly propositional in the theoretical complexities shown by the numbers of states: TRANS is much faster than 3-SP; ratios of computational times between TLEN and others are relatively small in comparison with ratios of numbers of states. Moreover, computational times depend on combinations of queries and graphs. For example, the theoretical inefficiency of TLEN comes to the surface when there are few shortcutting routes because of the necessity to consider paths of many edges.

Someone may notice that a 3-SP problem can be solved by a composition of two SP queries: When we want to find a shortest route from a vertex v_1 to a vertex v_2 via a vertex v_3, it is sufficient to find the shortest paths from v_1 to v_3 and from v_3 to v_2. However, 3-SP is about three times slower than SP on our system. Therefore, there are rooms for further optimization.

6. Related Works

6.1 Variants of Shortest Path Problems

There are many studies about variants of shortest path problems, such as additional constraints and variation of costs (Joksch 1966; Desrochers and Soumis 1988; Romeuf 1988; Punnen 1991; Barrett et al. 2000; Brodal and Jacob 2004; Villeneuve and Desaulniers 2005; Sherali et al. 2006). Optimal path querying systems were also proposed. For example, regular-language-constrained shortest path queries on a time-dependent network are available on the route planner of TRANSIMS system (Barrett et al. 2002, 2007). We aimed to give a general framework that includes many of them. However, there are still several problems that cannot be dealt with in our framework even though efficient algorithms are known, for example maximum capacity path problems (Punnen 1991) and context-free-language-constrained shortest path problems (Barrett et al. 2000).

In general, we can solve constrained shortest path problems based on ranking shortest path algorithms (Martins 1984; Eppstein 1998): enumerate paths from shorter ones until a feasible path is found. Although this procedure works for any constrained shortest path problems, even termination of the procedure is hard to guarantee.

Chan and Zhang (2007) proposed a generic optimal path querying algorithm on spatial databases. What their algorithm can deal with is, intuitively, a problem satisfying Bellman's principle of optimality (Bellman 1957): each subpath of optimal paths must be optimal. While our framework does not require the property, it seems difficult to implement our framework on spatial databases.

[2] 9th DIMACS Implementation Challenge - Shortest Paths. 2006. `http://www.dis.uniroma1.it/~challenge9/`.

Since we reduced optimal path problems into shortest path problems, known results about shortest path problems would be useful for our framework. For instance, while we use Dijkstra algorithm to find an optimal path, use of A* search algorithms or the bidirectional Dijkstra search algorithm would improve efficiency. Ranking shortest path algorithms (Martins 1984; Eppstein 1998) might be also useful to obtain nearly-optimal results.

6.2 Derivation of Optimal Path Querying Algorithm

Our derivation of the optimal path querying algorithm consists of three parts: designing a language from which we can obtain sufficient information of case analyses; reducing optimal path problems into shortest path problems; constructing an efficient graph searching procedure based on by-need evaluation.

The design of our language is highly influenced by the work of Sasano et al. (2000). They gave a generic solution for a class of optimization problems on a tree, called *maximum marking problems*, where recursive functions are used to specify problems. In our previous work (Morihata et al. 2007), we generalized the result of Sasano et al. and showed derivations of algorithms for regular-language-constrained shortest path problems. In this paper, we gave a shape to the idea by designing a query language.

Ogawa et al. (2003) also gave a framework to query and analyze graphs efficiently based on recursive functions. Their work requires the underlying graph to be k-tree decomposable and users to write recursive function on the structure of tree decompositions. Such requirements make the use of the framework hard.

Neither the use of product construction nor by-need evaluation is our new invention. Romeuf (1988) showed that regular-language-constrained shortest path problems can be reduced into shortest path problems by product construction. Similar approaches were also adopted by de Moor et al. (2003) and Liu et al. (2004) to obtain algorithms for regular path queries. Barrett et al. (2002) and Sherali et al. (2006) used by-need evaluation in their optimal path querying algorithms.

We derived an algorithm based on finite state automata. In fact, our derivation follows classic results about correspondence among finite state automata, dynamic programming, and shortest path problems. Karp and Held (1967) showed that finite state automata gave good characterization of dynamic programming algorithms. Ibaraki (1973, 1978) extended their results, and showed that once a problem is specified in terms of a finite state automaton, we can solve the problem by algorithms for shortest path problems.

7. Conclusion

In this paper, we have proposed a query language for optimal path problems. The use of recursive functions enable us to describe a wide range of problems. We have shown a derivation of an optimal path querying algorithm, in which recursive functions played an essential role. We also have implemented our idea as an optimal path querying system to evaluate our approach, which reveals it successful.

For further studies, it would be interesting to design a query language that is easier to write. For example, use of regular expressions or monadic second-order logic would be helpful to specify queries. Experiments with practical applications would be another interesting topic.

Acknowledgments

Authors are grateful to Zhenjiang Hu for his advice to study optimal path problems, Nanao Kita for her careful proofreading of our early draft, JSSST-SIG-PPL members for discussions about improvement of our system, and anonymous referees of ICFP 2008 for their helpful comments. The first author is supported by the Grant-in-Aid for JSPS research fellows $20 \cdot 2411$.

References

Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30 (3):809–837, 2000.

Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2002.

Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Label constrained shortest path algorithms: An experimental evaluation using transportation networks. Technical report, Virginia Tech (USA), Arizona State University (USA), and Karlsruhe University (Germany), 2007.

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004.

Edward P. F. Chan and Jie Zhang. A fast unified optimal route query evaluation algorithm. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 371–380. ACM, 2007.

Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2):15–35, 2003.

Martin Desrochers and François Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFOR*, 26: 191–212, 1988.

David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

Sergio Flesca, Filippo Furfaro, and Sergio Greco. Weighted path queries on semistructured databases. *Information and Computation*, 204(5):679–696, 2006.

Toshihide Ibaraki. Solvable classes of discrete dynamic programming. *Journal of mathematical analysis and applications*, 43(3):642–693, 1973.

Toshihide Ibaraki. Branch-and-bound procedure and state-space representation of combinatorial optimization problems. *Information and Control*, 36(1):1–27, 1978.

Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

H. C. Joksch. The shortest route problem with constraints. *Journal of Mathematical analysis and applications*, 14:191–197, 1966.

Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

Turgay Korkmaz and Marwan Krunz. Multi-constrained optimal path selection. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 834–843, 2001.

Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 219–230. ACM, 2004.

Ernesto Q. Vieira Martins. An algorithm for ranking paths that may contain cycles. *European Journal of Operational Research*, 18(1):123–130, 1984.

Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Jornal on Computing*, 24(6):1235–1258, 1995.

Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Calculus of minimals: Deriving dynamic-programming algorithms based on preservation of monotonicity. *Technical Report METR* 2007-61, Department of Mathematical Informatics, University of Tokyo, 2007.

Mizuhito Ogawa, Zhenjiang Hu, and Isao Sasano. Iterative-free program analysis. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP'03, Uppsala, Sweden, August 25-29, 2003*, pages 111–123. ACM, 2003.

Abraham P. Punnen. A linear time algorithm for the maximum capacity path problem. *European Journal of Operational Research*, 53(3):402–404, 1991.

Jean-François Romeuf. Shortest path under rational constraint. *Information Processing Letters*, 28(5):245–248, 1988.

Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 137–149. ACM, 2000.

Hanif D. Sherali, Chawalit Jeenanunta, and Antoine G. Hobeika. The approach-dependent, time-dependent, label-constrained shortest path problem. *Networks*, 48(2):57–67, 2006.

Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2001.

Daniel Villeneuve and Guy Desaulniers. The shortest path problem with forbidden paths. *European Journal of Operational Research*, 165(1): 97–107, 2005.

A. Proofs of Lemmas

Here we give proofs of lemmas. Note that we regard each expression in the description as a function that takes a path and returns a value.

LEMMA A.1. For each boolean-valued expression ϕ in the description, $\phi(x \cdot a) = \phi(y \cdot a)$ holds if $state(x) = state(y)$ holds.

Proof . Inequalities are the only nontrivial construction. We will proof it by contradiction.

Assume that $e(x \cdot a) \leq n < e(y \cdot a)$ holds for an inequality "$e \leq n$". Then, there exists an integer-valued function $g \rightsquigarrow e$ such that $g(x) < g(y)$ holds. Since $state(x) = state(y)$ holds, $g(x) < g(y)$ implies $\hat{g}(x) = \hat{g}(y) = \infty$. From the definition of $\hat{g}$ and the fact $g \rightsquigarrow e$, $\hat{g}(x) = \infty$ implies $g(x) > n$, which contradicts $e(x \cdot a) \leq n$. □

LEMMA A.2 (Lemma 4.1). If $state(x) = state(y)$ holds for two sequences x and y, then $state(x \cdot a) = state(y \cdot a)$ holds for any edge a.

Proof . From Lemma A.1, $p(x \cdot a) = p(y \cdot a)$ holds for all boolean-valued functions p in a description; thus, it is sufficient to show $\hat{f}(x \cdot a) = \hat{f}(y \cdot a)$ holds for all integer-valued functions f in a description. We will proof it by contradiction.

Assume that $\hat{f}(x \cdot a) < \hat{f}(y \cdot a)$ holds. From Lemma A.1, the same branches are chosen at all conditional expressions in the computations of $\hat{f}(x \cdot a)$ and $\hat{f}(y \cdot a)$. Therefore, there exists an integer-valued function $g \rightsquigarrow f$ such that $g(x) < g(y)$ and g is certainly called when $\hat{f}(x \cdot a)$ and $\hat{f}(y \cdot a)$ are evaluated. Since $state(x) = state(y)$ holds, $g(x) < g(y)$ implies $\hat{g}(x) = \hat{g}(y) = \infty$. Now let u be the boundary used in $\hat{f}$; then, $\hat{g}(x) = \infty$ implies $g(x) > u$, because $f \overset{+}{\rightsquigarrow} e$ implies $g \overset{+}{\rightsquigarrow} e$ for any expression e. However, $g(x) \leq f(x \cdot a)$ holds from construction of f, which implies $u < f(x \cdot a)$. In summary, $\hat{f}(x \cdot a) = \infty$ holds and it contradicts $\hat{f}(x \cdot a) < \hat{f}(y \cdot a)$. □

LEMMA A.3 (Lemma 4.2). For a edge a and two paths x and y such that $pstate(x) = pstate(y)$ holds, $x \cdot a$ is a feasible path if and only if $y \cdot a$ is a feasible path.

Proof . From the definition of $pstate$, $dst(x) = dst(y)$ holds; hence $x \cdot a$ is a path if and only if $y \cdot a$ is a path. Moreover, $state(x) = state(y)$ holds; hence, from Lemma A.1, $x \cdot a$ is feasible if and only if $y \cdot a$ is feasible. □

LEMMA A.4 (Lemma 4.3). For the objective function h and two sequences x and y, assume that both $state(x) = state(y)$ and $h(x) \leq h(y)$ hold. Then, for any edge a, both $h(x \cdot a) \leq h(y \cdot a)$ hold.

Proof . The value of $h(x \cdot a)$ is determined from the value of recursive function calls and values determined from a. Recall that the body of h includes no function calls except for that of h. Moreover, from Lemma A.1, the same branches are chosen to evaluate $h(x \cdot a)$ and $h(y \cdot a)$. Now the difference of $h(x \cdot a)$ and $h(y \cdot a)$ comes from only the value of $h(x)$ and $h(y)$. Since $h(x) \leq h(y)$ holds from an assumption, we can conclude $h(x \cdot a) \leq h(y \cdot a)$ from construction of h. □

LEMMA A.5 (Lemma 4.5). If $L_{\mathcal{S}'(state(x))} = \emptyset$ holds for a sequence x, $x +\!\!+ y$ is not feasible for any sequence y.

Proof . It is evident because $y \in L_{\mathcal{S}'(state(x))}$ is equivalent to that $x +\!\!+ y$ is feasible if we ignore the labels. □

LEMMA A.6 (Lemma 4.6). For the objective function h and two sequences x and y, assume that all of $L_{\mathcal{S}'(state(x))} \supseteq L_{\mathcal{S}'(state(y))}$, $dst(x) = dst(y)$, and $h(x) \leq h(y)$ hold; then, for any sequences z, $x +\!\!+ z$ is feasible if $y +\!\!+ z$ is feasible, and $h(x +\!\!+ z) \leq h(y +\!\!+ z)$ holds.

Proof . As similar to the case of Lemma 4.5, it is evident that $L_{\mathcal{S}'(state(x))} \supseteq L_{\mathcal{S}'(state(y))}$ implies that $x +\!\!+ z$ is feasible if $y +\!\!+ z$ is feasible.

Consider $z' \in L_{\mathcal{S}'(state(y))}$, and let z be an sequence obtained by removing labels from z'. Since $z' \in L_{\mathcal{S}'(state(x))}$ holds, the branches used for computing $h(x +\!\!+ z)$ from $h(x)$ is the same as that for $h(y +\!\!+ z)$ from $h(y)$. Therefore, $h(x +\!\!+ z) \leq h(y +\!\!+ z)$ holds from the construction of h, as similar to the proof of Lemma A.4. □