A Generic C++ Library for Solving Path Problems

Matko Botinčan
Department of Mathematics, University of Zagreb, Croatia

Path problems are a family of frequently encountered optimization and enumeration problems. Although they are specific each for itself, theory provides a mathematical framework for treatment of path problems in a general way. In this paper we describe a generic C++ library based on algebraic approach for solving path problems. The classes and functions in the library are very compact and written with the intention to be extensively combined. In this way, many different concrete path problems can be solved by using the same set of programming components.

Keywords: path problems, algebraic approach, semirings, generic programming, C++ programming language.

1. Introduction

Path problems, in a broad sense, can be seen as a general name for various kinds of problems originating from different fields. They are usually connected to areas of operational research and computer science where determination and evaluation of paths in directed graphs is a commonly encountered problem.

Probably the best known example is the shortest path problem where, in a directed graph with assigned lengths of arcs, the task is to determine the shortest path between two given nodes. A similar kind of problem is to find the longest path, the most reliable path or, alternatively, a path with maximum capacity. There are also examples of enumeration problems such as checking the existence of a path between two given nodes or listing all possible paths. Problems from other areas, such as determination of language accepted by a finite automaton, flow analysis of computer programs or some graph-theoretic problems, can also be treated as instances of path problems.

Single instance of a path problem can be solved by dedicated algorithms, but separately from other path problems. Our aim is to present a general framework for dealing with all path problems simultaneously, enabling in that way application of general algorithms. As it will be shown in the next section, with the use of algebraic approach to path problems, we shall accomplish specified goals.

Algebraic approach can seem very attractive from the theoretical point of view as it enables abstraction of problem specific details and treats the core of the path problems in a unifying way. Although there are many interesting theoretical issues concerning algebraic approach, in this paper we shall point out usefulness of practical application of such method.

Algebraic approach will be used as a basis for design and implementation of a small, but very flexible and applicable generic C++ library of classes and functions for dealing with path problems. Use of such library will provide programming environment for elegant treatment of path problems and solving them in a general way.

The paper is organized as follows. Section 2 reviews the theory from [1] and [4] and gives a quick overview of the algebraic approach to path problems. At the end of the section some examples will also be shown. Section 3 describes design and implementation of C++ classes and functions needed for realization of underlying algebraic structures. Section 4 presents implementation of basic iterative algorithms for solving path problems that are designed to work with implemented algebraic framework. In the final Section 5 concluding remarks will be given.

2. Algebraic Approach

Several approaches for solving general path problems can be found in literature. We use algebraic approach developed in [4], which treats given path problem as a system of equations defined in a suitable chosen algebraic structure. In that sense, finding a solution to a path problem will actually mean finding a solution to a system of equations.

The algebraic structure which has been recognized as fundamental in path problems is a *semiring* $(S, \oplus, \otimes)$ — a set S with two binary operations $\oplus$ (addition or join) and $\otimes$ (multiplication) with the following properties: $(S, \oplus)$ is a commutative additive monoid with zero element $\mathbb{0}$: $x \oplus y = y \oplus x$, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, $x \oplus \mathbb{0} = x$; $(S, \otimes)$ is a multiplicative monoid with unit element $\mathbb{1}$ and absorbing element $\mathbb{0}$: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$, $x \otimes \mathbb{1} = \mathbb{1} \otimes x = x$, $x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$; operation $\otimes$ is left and right distributive over $\oplus$: $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$, $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$.

For example, $(\mathbb{R} \cup \{\infty, -\infty\}, \min, +)$ semiring presents the underlying algebraic structure for shortest path problem in directed graphs: $\oplus$ denotes min, $\otimes$ denotes $+$, $\mathbb{0}$ denotes ∞ and $\mathbb{1}$ denotes 0. Another example is boolean semiring $(\{0, 1\}, \max, \min)$ that arises in path existence problem. It consists of the set $\{0, 1\}$ and max and min as binary operations $\oplus$ and $\otimes$ having 0 as zero and 1 as unit element.

According to notation used in algebra, we shall designate the product of an element of S with itself by the power notation: $x^n = \bigotimes_{k=1}^{n} x = x \otimes x \otimes \cdots \otimes x$ (n times). Algebraic closure of the element x is denoted by x^* and is defined by infinite sum $x^* = \bigoplus_{k=0}^{\infty} x^k = \mathbb{1} \oplus x \oplus x^2 \oplus x^3 \oplus \cdots$. Semirings $(S, \oplus, \otimes)$ in which x^* exists for every x in S are called closed semirings. Hereafter we shall assume that all observed semirings are closed.

A semiring $(S, \oplus, \otimes)$ having partial order relation $\leq$, which is monotone with respect to both operations $\oplus$ and $\otimes$, is called an ordered semiring. All natural examples of ordered semirings that are found in applications are ordered by difference relation (or naturally ordered): $x \leq y$ if and only if there exist $z \in S$ such that $x \oplus z = y$.

It is often the case that semiring being observed is idempotent, i.e. idempotent law holds for $\oplus$: $x \oplus x = x$. For this class of semirings operation $\oplus$ generates partial order on S by relation: $x \leq y$ if and only if $x \oplus y = y$.

Let $M_n(S)$ denote the set of all $n \times n$ matrices over semiring $(S, \oplus, \otimes)$. $M_n(S)$ forms another semiring if matrix addition and multiplication are defined just as usual in linear algebra in terms of "scalar" operations $\oplus$ and $\otimes$. Zero element is the zero matrix whose all entries are equal to $\mathbb{0}$, and the unity element is unit matrix (denoted by I) having $\mathbb{1}$'s on the main diagonal and $\mathbb{0}$'s elsewhere. Powers of the matrix are defined analogously as in the "scalar" case.

It can be shown ([4]) that one natural way to define path problem in the semiring algebraic structure is by a system of equations being written in a matrix form as:

$$X = I \oplus AX, \tag{1}$$

where, for a given matrix $A = [a_{ij}]$, unknown elements of the matrix $X = [x_{ij}]$ have to be found. Depending on the chosen semiring, interpretation of the elements of matrices A and X may vary. For example, in the case of the shortest path problem in directed graphs, matrix A usually represents adjacency matrix for a graph, whereas elements of matrix X are unknown lengths of the shortest path between two nodes in a graph.

It can easily be seen that the formal solution of equation (1) can always be found in terms of the algebraic closure of the matrix A: $A^* = \bigoplus_{k=0}^{\infty} A^k = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$. If this sequence remains stable after a finite number of iterations (in such case we say the matrix A is stable), the obtained sum actually represents a solution of (1). In all meaningful path problems matrix A is stable, i.e. the preceding sum becomes saturated after adding finite many powers of A. Thus, in general, we shall denote the solution (or some solution) of (1) with A^*.

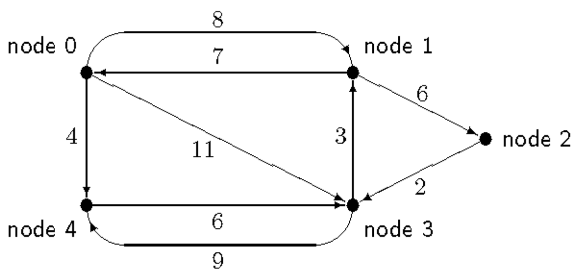

Fig. 1. A sample directed graph with given lengths of arcs.

To illustrate mentioned ideas, let us consider an example of a path problem for the directed graph shown in Figure 1. The task is to solve the shortest distance problem for the given graph, i.e. to determine the length of a shortest path between any two nodes in the graph. As it was mentioned, for this particular problem semiring $(\mathbb{R} \cup \{\infty, -\infty\}, \min, +)$ will be chosen. As the matrix A should represent adjacency matrix for a given graph, its element a_{ij} will be weight of the edge between nodes i and j (by defining the weight to be ∞ where the edge between two nodes does not exist). In our example matrix A looks as:

$$A = \begin{bmatrix} \infty & 8 & \infty & 11 & 4 \\ 7 & \infty & 6 & \infty & \infty \\ \infty & \infty & \infty & 2 & \infty \\ \infty & 3 & \infty & \infty & 9 \\ \infty & \infty & \infty & 6 & \infty \end{bmatrix}$$

The solution of the path problem equation (1) (where in our case unknowns x_{ij} represent lengths of the shortest paths between nodes i and j) is the matrix A^*:

$$A^* = \begin{bmatrix} 0 & 8 & 14 & 10 & 4 \\ 7 & 0 & 6 & 8 & 11 \\ 12 & 5 & 0 & 2 & 11 \\ 10 & 3 & 9 & 0 & 9 \\ 16 & 9 & 15 & 6 & 0 \end{bmatrix}$$

The matrix A^* is obtained by computing a closure of the matrix A. Note that A is indeed stable since the corresponding graph does not contain a cycle of negative length.

As a second example of path problem for graphs, let us consider again the same graph from Figure 1, but assume now that the task is to solve path existence problem, i.e. to determine which pairs of nodes are connected by a path and which are not. Now we can use much simpler algebraic structure — boolean semiring $(\{0, 1\}, \max, \min)$. The corresponding matrices A and A^* have the same structure regarding zero/nonzero elements as in the first example, but any ∞ is now replaced with 0 and all other values with 1. The (i, j)-th element of the matrix A indicates the existence of an arc between nodes i and j, while the (i, j)-th element of the matrix A^* designate the existence of a path from node i to node j.

3. Implementation of Algebraic Structures

The main purpose of the C++ library that is to be described is to provide a generic programming framework for solving path problems in general way. Ideas of algebraic approach presented in the previous section will be used as a guide for design and implementation of the library.

Algebraic approach consists of two basic steps — interpretation of a given path problem with a suitably chosen algebraic structure and solving obtained system of equations by a generally applicable algorithm. In this section our goal is to show how to effectively and elegantly represent these underlying algebraic structures by using some of syntactic advantages of the C++ programming language ([5]).

Following the results from the preceding section, algebraic structure that is found to be fundamental for solving path problems is semiring. From the programmer's point of view, it is a type for which binary operations of addition and multiplication with adequate properties have to be defined. In that manner, we shall start description of our C++ library with classes which represent respective binary operations.

Due to reasons that will become apparent later, our choice for representation of binary operations is to use classes that implement static function (ßapply()) for applying binary operation on elements of a given set and have static member variable (ßneutral) that provides neutral element for implemented binary operation. In our library we provide several generic classes which implement commonly used binary operations. For example, a class that represents a binary operation of using plus operator on elements of some type looks as following:

```
template <class T>
class PlusOp {
public :
    static T neutral;
    static T apply(const T& x,
        const T& y) {
        return x + y;
    }
};
```

Note that as PlusOp<T>::neutral is a static member of class PlusOp, we must define it somewhere outside the class. As the most common neutral element for the plus operator is zero, a generic definition can be provided:

```
template <class T>
T PlusOp<T>::neutral = 0;
```

Another example is a class that represents binary operation of taking minimum of two elements:

```
template <class T>
class MinOp {
public :
    static T neutral;
    static T apply(const T& x,
        const T& y) {
        return x < y ? x : y;
    }
};
```

As it was the case with PlusOp's static member, we must give a definition of MinOp<T>::neutral. Unfortunately, it is hard to give generic definition because its definition differs even for built-in types. For real types such as **float** or **double**, that have representation of positive infinity, we can define neutral element by using numeric_limits<T>::infinity() from standard C++ library:

```
template <>
double MinOp<double >::neutral =
    std::numeric_limits<double >::
    infinity();
```

For integral types which do not have representation of infinity, we can simulate the same by using a maximum finite value for a type. In the case of the type **int**, definition would look as following:

```
template <>
int MinOp<int >::neutral =
    std::numeric_limits<int>::
    max();
```

Note that in this case we now have to specialize generic definition of function PlusOp<T>::apply() for type **int** in order to incorporate desired behavior regarding MinOp<**int** >'s chosen neutral element:

```
template <>
int PlusOp <int >::apply(const int & x,
    const int & y) {
    static int INF =
        std::numeric_limits<int >::max()/2;
    if (x >= INF || y >= INF)
        return INF;
    else
        return x + y;
}
```

In general, when generic definitions of functions and classes proposed in the library are not satisfactory, a user can provide specialized versions that will deal with particularities regarding an assigned type. We should mention that the presented binary operation classes are meant to be used just as function wrappers, thus they differ from the standard notion of C++ function objects. The reason for choosing such implementation is to minimize run-time overhead when invoking particular binary operations.

All significant properties of the semiring algebraic structure can now be implemented through the generic class Element. The class is parameterized by one template argument presenting the type of the elements (i.e. underlying set for semiring) and two template arguments through which binary operations of addition and multiplication are specified. The class provides overloaded versions of arithmetic operators which will be used for invoking respective binary operations. The most significant part of the class looks as following:

```
template <class T,
    template <class > class U,
    template <class > class V>
class Element {
    T val;
public :
    typedef U<T> Plus;
    typedef V<T> Times;
    Element() {
    }
    template <class X>
    Element(const X& x) : val(x) {
    }
    operator T() {
        return val;
    }
    Element operator +(const Element& x)
        const {
        return Plus::apply(val, x.val);
    }
    Element operator *(const Element& x)
        const {
        return Times::apply(val, x.val);
    }
    ...
};
```

Definitions of the other overloaded arithmetic operators in the class Element are left out as they are implemented in the same way.

As it can be seen from the definition of the class Element, sizeof(Element<T>)==sizeof(T) for every type T, there is no space overhead against type T. The cast operator to type T is also provided, in order to enable the use of instances of Element<T> in almost all contexts as instances of T.

4. Implementation of Algorithms

In this section it remains to show how to combine the developed programming elements with general algorithms for solving systems of equations in order to solve the algebraic formulation of a given path problem. In our library we have implemented algorithms for solving two types of path problem: all-pairs problem, which refers to computing a solution of (1), and single-source problem where the goal is to find only one row or one column of the unknown matrix X.

To find a solution of both types to path problem, direct solution procedures (elimination algorithms) can be used. They are based on successive elimination of variables that are very similar to those used in linear algebra for solving ordinary systems of linear equations. Probably the most well-known algorithm using this kind of approach are Gauss-Jordan eliminations. Divide and conquer strategies can also be used, thus leading to recursive algorithms (i.e. escalator method).

As it is the case in the classical numerical linear algebra, another approach for computing a solution of a system of equations based on iterative solution procedures can be used. When the matrix semiring is closed and idempotent, iterative algorithms are easy to write down, thus for the purpose of this paper, that case will be presented.

The basic iterative algorithm for solving all-pairs path problems is actually based on computing closure A^* of a given matrix A. The defining expression of matrix closure can be evaluated relatively quickly by successive squaring of the matrix. More precisely, the following sequence of matrices is computed:

$$\begin{aligned} C_0 &= I \oplus A \\ C_k &= C_{k-1} \otimes C_{k-1} \quad (k = 1, 2, \ldots) \end{aligned}$$

Computation is terminated when for some k matrix C_k becomes equal to matrix C_{k-1} (note that it will always happen since we assumed idempotence of matrix semiring). It is easy to verify that at that point matrix C_k equals to closure matrix A^*.

We present implementation of a slight variation of this algorithm where the successive elements of matrix C in each iteration are always computed from its newest values (it can be shown that in the case of idempotent semirings this modification preserves correctness of the algorithm).

```
template <class Elem, class Matrix1,
    class Matrix2>
void Closure(const Matrix1& A, int n,
    Matrix2& C) {
    int i, j, k;
    bool ok = true ;
    Elem t;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            C[i][j] = A[i][j] + (i==j ?
                Elem::Times::neutral :
                Elem::Plus::neutral);
    while (ok) {
        ok = false ;
        for (i = 0; i < n; ++i)
            for (j = 0; j < n; ++j) {
                t = C[i][j];
                for (k = 0; k < n; ++k)
                    t += C[i][k]*C[k][j];
                if (!(t==C[i][j])) {
                    C[i][j] = t;
                    ok = true ;
                }
            }
    }
}
```

Analogously, for solving single-destination problems, a version of Gauss-Seidel iterative algorithm can be used, having the following implementation:

```
template <class Elem, class Matrix,
    class Vector>
void GaussSeidel(const Matrix& A,
    int n, int k, Vector& y) {
    int i, j;
    bool ok = true ;
    Elem t;
    for (i = 0; i < n; ++i)
        y[i] = A[i][k] + (i==k ?
            Elem::Times::neutral :
            Elem::Plus::neutral);
    while (ok) {
        ok = false ;
        for (i = 0; i < n; ++i) {
            t = y[i];
            for (j = 0; j < n; ++j)
                t += A[i][j]*y[j];
            if (!(t==y[i])) {
                y[i] = t;
                ok = true ;
            }
        }
    }
}
```

To conclude the section, we shall give examples of using the described C++ classes and functions for solving two sample path problems presented in Section 2. In case of the shortest distance problem for a directed graph, we would define the elements to be of the following type:

```
typedef
    Element<double , MinOp, PlusOp>
    element;
```

Assuming that A is a given adjacency matrix, C is a matrix where the solution (closure of A) will be stored and n is the dimension of both matrices, the function Closure() should be invoked as following:

```
Closure<element>(A, n, C);
```

If we would like to compute single-destination problem for A, a call to GaussSeidel() function will perform the desired task (here y presents a vector in which the computed solution will be stored and k indicates which column of the all-pairs problem is to be computed):

```
GaussSeidel<element>(A, n, k, y);
```

In the case when the path existence problem needs to be solved, the whole procedure of calling functions Closure() or GaussSeidel() remains essentially the same, only the type of the elements has to be defined as:

```
typedef
    Element<bool , MaxOp, MinOp>
    element;
```

5. Conclusion

In this paper we have described a generic C++ library of classes and functions for solving path problems in a general way. The established programming framework is essentially based on using algebraic approach to path problems which exhibits much wider software engineering principle of separation of algorithms and data and considerably increases reusability of the code.

Efficiency of the implemented algorithms is generally inferior to specialized algorithms used for solving particular path problems due to differences in their time complexities. Nevertheless, the intention of described library is to be used as a support for fast prototyping, testing and experimentation. Moreover, since the fundamental data structure behind all algorithms are matrices, possibility for easy and direct parallelization should also not be disregarded.

References

[1] CARRÉ B., *Graphs and Networks*, Oxford: Oxford University Press; 1979.

[2] MANGER R., A library of subroutines for solving path problems, in: Hunjak T., Martic Lj., Neralic L., editors, *Proceedings of the 6th International Conference on Operations Research (KOI' 96)*; 1996 October 1–3; Rovinj, Croatia. Zagreb: Croatian Operational Research Society; 1996., p. 49–56.

[3] MANGER R., Solving path problems on a network of computers, *Informatica* 2002; 26:91–100.

[4] ROTE G., Path problems in graphs, *Computing Supplement* 1990; 7:155–189.

[5] STROUSTRUP B., *The C++ Programming Language*, Third Edition, Addison-Wesley, 1998.

Received: June, 2004
Accepted: June, 2004

Contact address:
Matko Botinčan
Department of Mathematics
University of Zagreb
Bijenička cesta 30
10000 Zagreb
Croatia
e-mail: `mabotinc@math.hr`

MATKO BOTINČAN is a postgraduate student at the Department of Mathematics, University of Zagreb where he received his B.S. in 2002. His current scientific interests include theoretical computer science, combinatorial optimization and operations research.