Computational and Complexity Results for an Interior Point Algorithm on Multicommodity Flow Problems

Anil P. Kamath [1] [2] Narendra K. Karmarkar [3] , K.G. Ramakrishnan [3] ,

Extended Abstract

1 Introduction

In this paper, we present some computational and complexity results for an Approximate Dual Projective (ADP) variant of the interior point algorithm [?] for solving the minimum cost Multicommodity Network Flow Problem (MCNF). The motivation for this work is to study the performance of the ADP algorithm for solving a special class of LPs arising in MCNF. In MCNF, there are k commodities and k source-sink pairs (s_i, t_i). The problem is to simultaneously transport, at minimum total cost, d_i units of the i^{th} commodity from s_i to t_i subject to the constraint that the total flow of the commodities through each edge is bounded by capacity $u(e)$ of the edge.

Formally, an instance of the problem consists of a directed graph $G = (,)$, and a set of commodities . For each commodity $i \in$ we are given a source-sink pair $(s_i, t_i) \in \times$ and a demand $d_i \in \mathcal{R}$. Each edge $e \in$ has a capacity $u(e)$ and costs $c^{(i)}(e)$ for each commodity $i \in$. A multicommodity flow f consists of flows $f^{(i)}(e)$ of the i^{th} commodity on edge e. Flow f has to satisfy the following constraints.

$$f^{(i)}(e) \geq 0 \quad \forall e \in, i \in \tag{1}$$

$$\sum_{i \in} f^{(i)}(e) \leq u(e) \ \forall e \in \tag{2}$$

$$\sum_{wv \in} f^{(i)}(wv) - \sum_{vw \in} f^{(i)}(vw) = \begin{cases} 0 & v \neq s_i \text{ and } v \neq t_i \\ -d_i & v = s_i \\ d_i & v = t_i \end{cases} \quad \forall v \in; \forall i \in \ . \tag{3}$$

The objective is to minimize the total cost of the flow f. The objective function is given by

$$\min \sum_{i \in} \sum_{e \in} c^{(i)}(e) f^{(i)}(e).$$

Multicommodity flow problems arise in several practical applications [?] such as design of communication and computer networks and optimal distribution of resources. Though the MCNF problem is a natural extension of the single-commodity flow problem, most of the combinatorial algorithms developed for the latter do not

[1]Department of Computer Science, Stanford University, Stanford, CA 94305 USA

[2]Research partially supported by US Army Office Research Grant DAAL-03091-G-0102

[3]AT&T Bell Laboratories, Murray Hill, NJ 07974 USA

readily extend to the MCNF problem. This is because many of the nice properties present in single-commodity flow problem do not apply to the multicommodity case. Hence most of the algorithms developed for MCNF are based on linear [?, ?] or non-linear programming [?, ?]. Recently, some combinatorial algorithms have been suggested [?] for finding *approximate* solutions to the MCNF problem. The complexity of these algorithms is exponential in the precision of the solution.

2 An Overview of the Approximate Dual Projective Algorithm

In this section, we give a brief overview of the Approximate Dual Projective (ADP) algorithm for linear programming. For a complete description of the algorithm see [?]. The ADP method is an approximation to the projective algorithm of Karmarkar [?] applied to the dual. The ADP method makes discrete approximations to the continuous central trajectory. The discrete approximation consists of two steps; the objective step, and the reciprocal-estimates-improvement (REI) step. Figure ?? depicts the ADP method. The objective step is the dual affine scaling step [?] that

-bb-error =

Figure 1: Pictorial View of the Approximate Dual Projective(ADP) Algorithm

improves the objective function. The REI step has the effect of bringing the iterate closer to the central trajectory. The name reciprocal estimates improvement comes from the fact that on the central trajectory of the dual polytope, the reciprocals of slack variables are exact primal feasible solutions up to a multiple [?]. Thus if we take a step in the dual polytope that improves the primal feasibility of the reciprocals of slacks, then this necessarily implies that the step brings the iterate closer to the central trajectory. The ADP algorithm is similar to the BLCA algorithm of Todd [?], the affine scaling algorithms of Gonzaga [?],Barnes, Chopra, and Jensen [?], and many others.

3 Complexity of ADP Algorithm for MCNF

The computationally intensive step in each iteration of the ADP algorithm is the solution of a linear system of equations

$$AD^2A^Tx \;=\; b, \tag{4}$$

where A is the constraint matrix of MCNF, given by ?? to ?? and D is a diagonal scaling matrix. In our implementation, we use Preconditioned Conjugate Gradient (PCG) to solve the linear system. The Conjugate Gradient (CG) method is an iterative method which has been shown to be effective in solving large sparse positive definite linear systems. The CG method is guaranteed to converge in m steps, where m is the rank of the matrix, and each step involves multiplying the matrix by a vector [?].

Consider an instance of the MCNF problem, with k commodities, v vertices and e edges. The LP formulation of this instance has ke variables and $kv + e$ constraints ?? to ??. The linear system ?? has rank $kv + e$ and so the CG is guaranteed to terminate in $O(kv+e)$ steps. The constraint matrix A has $O(ke)$ non-zeros, and hence one CG step which involves computing AD^2A^Tx can be done in $O(ke)$ operations. Thus the number of operations per ADP iteration is $O(ke(kv+e))$. The ADP variant of the interior point method for LP has been shown to terminate in $O(\sqrt{n}L)$ iterations where n is the number of variables in the primal problem and L is the number of bits required to encode the problem [?]. Hence the overall complexity of ADP algorithm for MCNF can be shown to be $O(ke(kv + e)\sqrt{ke}L)$.

Better complexity for solving MCNF problem has been reported by Vaidya [?]. Vaidya's method uses fast matrix multiplication with direct factorization that uses the special nature of the matrix to solve the linear system [?]. However, for problem sizes of practical interest fast matrix multiplication is inefficient. This is because the constant multiplicative factors in the complexity measure for fast matrix multiplication are large. For sparse graphs, the complexity of our method matches the best known complexity as reported in [?].

4 Efficient Implementation of ADP Algorithm for MCNF

The constraint matrix in the LP formulation of MCNF has a block of conservation constraints for each commodity coupled by the capacity constraints on the edges. We store the constraint matrix implicitly by just storing a single copy of the incidence matrix of the underlying graph. All matrix operations on the constraint matrix are carried out by appropriate indexing in the incidence matrix. The capacity constraints are not stored since their structure is known.

We implemented the ADP algorithm for MCNF in an object-oriented fashion. We define a C++ class called MCF-matrix with member functions to do various operations on the matrix object. The software structure of our implementation is outlined in Figure f.objct. The outermost layer in the figure represents the main

-bb-error =

Figure 2: Object Oriented Structure of the Software

ADP algorithm. The implementation of the main algorithm is still in fortran-77 but it is matrix-transparent i.e., it has no knowledge of how and where the constraint matrix is stored. The interface to the C++ object is through the C++-Fortran interface as shown in the middle layer. The kernel of the software is the MCF-matrix object. This object has three subclasses, namely, the network object, the mask object and the commodities object. A pseudo C++ class definition is given below.

```
class MCF_matrix{
    network graph;
    mask maskobj;
    int commodities;
    public:
    vector ax(const vector x);
    vector atx(const vector x);
    vector adatx(const vector d,const vector x);
    matrix adat(const vector d);
    ⋮
};
```

The subclass *network* encodes the incidence matrix of the graph. The *mask* subclass is used for computing preconditioners. The information about commodities is stored in the scalar *commodities*. The publicly available member functions *ax , atx , adatx* are called by the C++-Fortran interface routines and return a vector object, for example, *ax* takes a vector x and multiplies it with the expanded MCF constraint matrix and returns the resulting vector. The function *adat* returns a matrix object. Additional member functions perform other related operations.

The motivation of this object-oriented implementation was to achieve compact storage, ease of vectorization, reduction in data access, and data transparency. Since

we store only a single copy of the incidence matrix, the storage required for representing the constraint matrix is reduced. The reduction in storage can be substantial for problems with large number of commodities. Each edge in the network subclass of our object representation corresponds to a vector of length $2k$ in the expanded constraint matrix, where k is the number of commodities. Thus, all matrix operations are readily vectorizable and the number of data accesses required to fetch the constraint matrix is reduced.. This data-hiding principle of object-oriented programming is central to our implementation. The main ADP algorithm is oblivious to the matrix representation and hence the same software can be reused for efficient implementation of other LPs by changing the object definition. For instance, we have reused the ADP software to solve concurrent flow problem, by doing minor modifications to the object.

5 Computational Experience

In this section, we describe the computational results for MCNF using our object oriented code for ADP and compare it with CPLEX-NETOPT [**?**]. All runs were carried out on a *Silicon Graphics Machine*, having a Risc-based architecture with 100MHZ clock. The ADP algorithm is implemented in FORTRAN and C++. The computational experiments were conducted on instances generated by two different generators. The first set of instances were generated using the *rmfgen* generator, described in [**?**]. An instance of the problem generated by *rmfgen*, denoted by $a.b.k$ has b layers of grids of size $a \times a$, and k commodities to be routed. In the instances we generated, the costs were generated randomly between 1 and 1000 and the capacities on the grid edges ranged from 1 to 10000. The demands varied randomly between 1 to 1000. The MCNF problems generated by *rmfgen* seem to be simple: we observed in our experiments that the sum of the optimal solutions to each of the single commodity flow problems was close to the optimal solution of the MCNF problem.

To generate difficult problems, we developed the *mgrid* generator. This generator creates a square 2-dimensional grid graph. Each edge in the grid is bidirectional and has a capacity associated with it in each direction. For each commodity we randomly choose a source on the boundary of the grid and the corresponding sink is chosen randomly on the opposite boundary. The demand for each commodity is generated independently at random. We solve a concurrent flow problem (CFP), in which we determine the maximum multiple μ of the demand vector that one can satisfy subject to the capacity constraints. The solution, μ, to the CFP problem is then used to scale the demands, thereby creating a feasible MCNF problem. We then randomly assign costs per unit flow for each commodity in each edge and solve the resulting MCNF problem. For the instances that we solved, the cost range was 1 to 1000 and the capacities ranged from 1 to 1000.

The performance of our ADP method and a comparison with the corresponding CPLEX-NETOPT times, for instances generated by *rmfgen* and *mgrid* are given in tables **??** , **??** respectively. The computational results for solving the CFP problems

Problem	Rows in A	Columns in A	ADP Time	CPLEX-NETOPT Time
16.8.4	17664	47360	283s	685s
16.8.8	25856	85248	505s	1253s
16.8.16	42240	161024	1441s	2300s
16.16.4	35584	96000	313s	2933s
16.16.8	51968	172800	1620s	5582s
16.16.16	84736	326400	6363s	11000s
16.32.4	71424	193280	1580s	12807s
16.32.8	104192	347904	10596s	23553s
16.32.16	169728	657152	19834s	44001s

Table 1: Performance of ADP on *rmfgen* problems

Grid Size	Comm.	Rows	Columns	ADP Time	CPLEX NETOPT
20	5	3520	9121	82s	515s
50	5	22300	58801	878s	2221s
100	5	89600	237601	8818s	$>$ 594533s

Table 2: Performance of ADP on the concurrent flow problem

that arise in the *mgrid* generator are reported in table ?? . All times reported are user times in seconds. Asymptotically, ADP performs better than the CPLEX-NETOPT runs and is nearly 40 times faster on some instances.

Grid Size	Comm.	Rows	Columns	ADP Time	CPLEX NETOPT
20	5	3520	9120	147s	408s
50	5	22300	58800	1740s	67956s
100	5	89600	237600	13102s	$>$ 556803s

Table 3: Performance of ADP on *mgrid* problems