November 29, 2006

RT0688

Computer Science; Mathematics 7 pages

# Research Report

Fast Shortest Path Computation for Solving the Multicommodity Flow Problem

Hiroki Yanagisawa

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

# Fast Shortest Path Computation for Solving the Multicommodity Flow Problem

Hiroki Yanagisawa
IBM Tokyo Research Laboratory
yanagis@jp.ibm.com

**Abstract**

For solving the multicommodity flow problems, Lagrangian relaxation based algorithms are fast in practice. The time-consuming part of the algorithms is the shortest path computations in solving the Lagrangian dual problem. We show that an A* search based algorithm is faster than Dijkstra's algorithm for the shortest path computations when the number of demands is relatively smaller than the size of the network.

**Keywords:** shortest path, A* search, multicommodity flow

## 1 Introduction

The multicommodity flow problem is to find the minimum cost flow which satisfies capacity constraints and flow conservation requirements while every pair of demands is delivered. Many logistics and communication network problems can be formulated as large multicommodity flow problems.

This problem can be easily formulated by linear programming, thus it is solvable in polynomial time. Many studies are done on this problem [1, 6] for solving large scale problem fast. At the best of my knowledge, Lagrangian relaxation based algorithm by Banonneau *et al.* [1] is the fastest in practice. While it mainly focuses on reducing the number of iterations when it solves the Lagrangian dual problem, it does not focus on accelerating the shortest path computations: It constructs *Dijkstra trees* (shortest path-spanning tree) by using Dijkstra's algorithm [3] for the shortest path computation. Since the shortest path computations are time-consuming [1] when we solve multicommodity flow problems, it is important to compute the shortest paths fast.

We show that an A* search [5] based algorithm is faster than Dijkstra's algorithm for computing the shortest paths when the number of demands is relatively smaller than the size of the network. When the number of demands is relatively small, it is inefficient to construct Dijkstra trees for computing the shortest paths of demands, because only a small fraction of the Dijkstra trees are utilized for computing the shortest paths of the demands. Therefore, it is better to compute the shortest paths of the demands separetely. While single source single target Dijkstra's algorithm can be used, A* search is a refined algorithm for this approach. A* search is originally designed for accelerating the shortest path computation in 1960's and the searches can be seen as *guided* search to the target node. Since A* search requires preprocessing, there was not much use for the shortest path computation (it was often used in the artificial intelligence area). Recently, because of attentions in these years on the P2P shortest path problem (a problem to find the shortest paths between the specified pairs of nodes), some studies [4] have been done on acceleration of the shortest path computaion by using A* search. Solving the Lagrangian dual problem is similar to solving the P2P shortest path problem in a sense that we compute the shortest path many times on the same network. Since the preprocessing time does not impact so much on the total execution time for solving the multicommodity flow problem, A* search will be suitable for solving the problem. In this paper, we show an empirical study of applying an A* search based algorithm to the multicommodity flow problem.

This paper is organized as follows. In section 2, we define the multicommodity flow problem. In section 3, we show an A* search based algorithm for the shortest path computation. In section 4, we show our results of empirical study. Section 5 concludes this paper.

## 2 The Linear Multicommodity Flow Problem

Given a network (graph) $G(N, A)$ with node set $N$ and arc set $A$, the linear multicommodity flow problem is defined as follows:

$$\begin{array}{ll} \min & \displaystyle\sum_{a\in A} c_a \sum_{k\in K} x_a^k \\ \text{subject to:} & \begin{array}{ll} \displaystyle\sum_{k\in K} x_a^k \le u_a, & \forall a \in A \\ Mx^k = d_k\delta^k, & \forall k \in K \\ x_a^k \ge 0, & \forall a \in A, \forall k \in K \end{array} \end{array}$$

where $M$ is the network matrix and $\delta^k$ is a vector of zeros except in the components associated with the end nodes of the demand of commodity $k$. $c_a$ is the unit cost on arc $a$ and $u_a$ is the capacity of arc $a$. $d_k$ is the demand for commodity $k \in K$. $x^k$ is variable for flow of commodity $k$ on the arcs of the network.

When we relax the capacity constraint, the Lagrangian dual problem is

$$\max_{\lambda \ge 0} L(\lambda)$$

where

$$L(\lambda) = \min_{x^k \ge 0, k\in K} \{L(x,\lambda) | Mx^k = \delta^k, \forall k \in K\},$$

and $L(x, \lambda)$ is the Lagrangian function

$$L(x,\lambda) = \sum_{a\in A} c_a \sum_{k\in K} x_a^k + \sum_{a\in A} \lambda_a \left( \sum_{k\in K} x_a^k - u_a \right).$$

The $L(\lambda)$ can be seen as a shortest path problem for the graph in which each arc $a$ has a cost $c_a + \lambda_a$.

## 3 Shortest Path Computation

A shortest path computation for a graph $G(N, A)$ is a fundamental problem in computer science. When we compute the shortest path from source node $s$ to target node $t$, Dijkstra's algorithm and A* search are often used.

### 3.1 Dijkstra's algorithm

Dijkstra's algorithm [3] is one of the fastest algorithm for the shortest path computation. It maintains *tentative distance* $d(v)$ for each node $v$ during execution. Initially, $d(s)$ is set to 0 for the source node $s$ and $d(v)$ is set to $\infty$ for node $v \ne s$. A priority queue stores *reached* nodes ($d(v) < \infty$) using $d(v)$ as the priority (initially the priority queue contains the source node $s$ only). In each iteration, the algorithm removes a node $u$ from the priority queue and scan the arcs coming out of $u$. To scan an arc $(u, v)$, check to see if $d(v) > d(u)+l(u, v)$, where $l(u, v)$ is a length (cost) of arc $(u, v)$. If so, we set $d(v) = d(u)+l(u, v)$

and put $v$ into the priority queue. When we reach the target node $t$, $d(t)$ is the shortest path cost from node $s$ to node $t$.

By executing the shortest path computation from a single node $s$ to all the other nodes, we can construct a Dijkstra tree whose root node is $s$.

### 3.2 A* search

A* search can be used for accelerating the shortest path computation by using preprocessing. It can be interpreted as defining *node potentials* $\pi_t(v)$ as lower bound of the distance $v$ to $t$ and corresponding *reduced arc weights* $l_\pi(u,v) = l(u,v) + \pi_t(v) - \pi_t(u)$ and solving it by Dijkstra's algorithm.

We call an algorithm *admissible* if it is guaranteed to find an optimal path from $s$ to $t$ for any graph and *feasible* if it is guaranteed to scan nodes in nondecreasing order of their distance from $s$ and scans each node at most once. To guarantee the admissibility and feasibility of the A* search, the node potentials must meets the following conditions.

**Theorem 3.1** *[5] If all the $\pi_t(v)$s give lower bounds of the shortest path costs $\forall v, t \in N$, then A* search is admissible.*

**Theorem 3.2** *[3, 4] If $l_\pi(u,v) \geq 0$ for all arcs $(u,v)$, then A* seach is feasible.*

The following theorem shows that better node potentials give better performance.

**Theorem 3.3** *[5, 4] Let $\pi_t$ and $\pi'_t$ be two feasible potential functions such that $\pi_t(t) = \pi'_t(t) = 0$ and $\pi'_t(v) \geq \pi_t(v)$ for any node $v$. Then the set of nodes scanned by A* search using $\pi'_t$ is a subset of the set of nodes scanned by A* search using $\pi_t$.*

Note that the above theorem implies that A* search with appropriate (admissible and feasible) node potentials scans no more nodes than Dijkstra's algorithm, since Dijkstra's algorithm is equivalent to the A* search with zero potential function.

### 3.3 A* search based algorithm

We show an A* search based algorithm here for solving the multicommodity flow problem. In the preprocessing stage of the algorithm, it solves the all pairs shortest path problem on graph $G(N,A)$ in which each arc $a \in A$ has a cost $c_a$ and it sets the shortest path cost from node $v$ to node $t$ as the node potentials $\pi_t(v)$. When it solves the Lagrangian dual problem, it uses A* search with the node potentials $\pi_t(v)$. Because each arc $a \in A$ has a cost $c_a + \lambda_a$ in graph $G(N,A)$ and $\lambda_a$ is nonnegative, the node potential $\pi_t(v)$ gives a lower bound of the shortest path costs from node $v$ to node $t$, which gurantees the admissibility. It is also easy to show that the node potential $\pi_t(v)$ guarantees the feasibility. Note also that the node potential $\pi_t(v)$ gives good lower bounds. Since some observations [1, 2] showed that the number of congested arcs in an optimal solution on practical problems is a small fraction of the total number of arcs in the network, most of node potential $\pi_t(v)$ is close to the shortest path cost from node $v$ to node $t$.

To improve the above A* search based algorithm, we make use of information of upper bounds of the shortest path costs. Suppose that we know in advance that the shortest path cost from $s$ to $t$ does not exceeds $UB$. During an execution of the A* search, $d(v) + \pi_t(v)$ gives a lower bound of the shortest path cost from $s$ via $v$ to $t$. If we find that $d(v) + \pi_t(v) > UB$ for a node $v$ during the A* search, we see that the shortest path from $s$ to $t$ does not go through $v$. Thus, we can skip putting $v$ into the priority queue. This improvement yields reducing the number of nodes in the priority queue, which accelerates the A* search. The pseudocode of the algorithm is given in Fig 1. The upper bounds are easily obtained while solving the Lagrangian dual problem. When we solve the Lagrangian dual problem, we iteratively calculates the shortest path costs for the same from-to pairs. So we memorize the shortest path in the previous iteration and, in the following iteration, we take an upper bounds as the total cost along the shortest path in memory.

```
Algorithm A* search($s$, $t$, $UB$)
    $s$ and $t$ are nodes and $UB$ is an upper bound of $s - t$ shortest path cost
begin
    forall $v \in N$ { $k(v) := \infty$; $d(v) := \infty$; }
    $k(s) := 0$; $d(s) := 0$;
    $Q := \{s\}$;
    while $Q$ is not empty
        delete $v$ from $Q$ such that $v := \arg_{v \in Q} \min k(v)$;
        forall $(v, w) \in A$
            if $k(w) > k(v) + l(v, w) - \pi_t(v) + \pi_t(w)$ then
                $k(w) := k(v) + l(v, w) - \pi_t(v) + \pi_t(w)$;
                $d(w) := d(v) + l(v, w)$;
                if $d(w) + \pi_t(w) \leq UB$ then { $Q := Q \cup \{w\}$; }
            endif
        endfor
    endwhile
end
```

Figure 1: A* search using an upper bound

Overall, the A* based algorithm is written as follows:

1. Set the lagrangian multipliers to zeros.
2. Calculate all pairs shortest paths for $G(N, A)$ and set them as node potenitals.
3. Update the lagrangian multipliers.
4. For each pair of demands, calculate upper bound (i.e. the total cost along the shortest path in the previous iteration).
5. For each pair of demands, calculate the shortest path by using the upper bound (by using algorithm in Fig. 1) and stores it.
6. Go to step 3.

For a further acceleration technique, we partition the demands into groups according to their destination nodes and we make use of the shortest paths in the same group. Suppose that the shortest path from node $s$ to node $t$ is given and that node $u$ is on the shortest path $s - t$ (i.e. the path $s - u - t$ is the shortest path). Since a subpath of the shortest path is also a shortest path, path $u - t$ is the shortest path from node $u$ to node $t$. When we compute the shortest path from node $s'$ to node $t$, we can use the fact that path $u - t$ is the shortest path. That is, when we reach node $u$ during the A* search from node $s'$ to $t$, we see that the path $s' - u - t$ is the shortest path among paths which goes through node $u$. Therefore, when we reach the node $u$, we scan all the edges on the shortest path $u - t$, instead of putting $u$ into the priority queue. That is, for each arc $(v, w)$ on the shortest path $u - t$, we remove node $w$ from the priority queue (if any), set $d(w)$ to the sum of $d(v)$ and the cost of the shortest path $v - t$ if necessary, and put $w$ into the queue. This acceleration technique improves execution time drastically.

## 4 Experiment

We conducted experiments of the A* search based algorithm against Dijkstra's algorithm. Since our focus is on accelerating the shortest path computations, we adopted a simple subgradient method for solving the Lagrangian dual problem.

### 4.1 Subgradient method

We used a subgradient method to obtain a solution. The subgradient $s_a(\lambda)$ shows subgradient of $L$ at $\lambda$ with respect to arc $a \in A$. The lagrangian multipliers $\lambda^i$s are updated by

$$\lambda_a^{i+1} = \lambda_a^i + d\frac{UB - L(\lambda^i)}{||s(\lambda^i)||^2} s_a(\lambda^i) \;\; \text{(for all } a \in A\text{)},$$

where $UB$ is an upper bound on the objective value of the original problem and $d$ is a step size parameter satisfying $0 < d \leq 2$. In our implementation, $UB$ is set to twice the maximum Lagrangian dual cost so far, and $d$ is initially set to 2 and halved whenever the lower bound does not increase in 20 consecutive iterations. The iteration is terminated after 500 iterations.

### 4.2 Results

Experiments are conducted on a PC (Pentium 4, 3.2 GHz, 3GB of RAM) under Windows XP operating system.

We used two sets of test problems: `planar` problems and `grid` problems. The `planar` problems are generated to simulate telecommunication problems and the `grid` problems has a grid structure such that each node has four incoming and four outgoing arcs. These two sets of problems are given in [1], which are originally given in [6].

Table 1 displays data on the two sets of problems. For each instance, we give the number of nodes $|N|$, the number of arcs $|A|$, the number of demands $|K|$, the objective value $z*$ of an optimal solution (given in [1]).

| Instance ID | $\lvert N\rvert$ | $\lvert A\rvert$ | $\lvert D\rvert$ | $z*$ |
|---|---|---|---|---|
| `planar300` | 300 | 1680 | 3584 | $6.89982 \times 10^8$ |
| `planar500` | 500 | 2842 | 3525 | $4.81984 \times 10^8$ |
| `planar800` | 800 | 4388 | 12756 | $1.16737 \times 10^8$ |
| `planar1000` | 1000 | 5200 | 20026 | $3.44962 \times 10^9$ |
| `grid10` | 625 | 2400 | 2000 | $1.64111 \times 10^8$ |
| `grid11` | 625 | 2400 | 3000 | $3.29259 \times 10^8$ |
| `grid12` | 900 | 3480 | 6000 | $5.77189 \times 10^8$ |
| `grid13` | 900 | 3480 | 12000 | $1.15932 \times 10^9$ |
| `grid14` | 1225 | 4760 | 16000 | $1.80268 \times 10^9$ |
| `grid15` | 1225 | 4760 | 32000 | $3.59353 \times 10^9$ |

Table 1: Test instances

We implemented Dijkstra's algorithm and the A* search based algorithm for comparison. We used binary heap to maintain the priority queue. In Dijkstra's algorithm, we partition demands according to their origin nodes and construct $|N|$ Dijkstra tree for each node. In the A* search based algorithm, we partition demands according to their destination nodes and the lower bounds (node potentials) are calculated by constructing Dijkstra tree for each node.

Table 2 shows the results. For each instance, we give the execution time of preprocessing of the A* search based algorithm, the total time for 500 iterations of the A* search based algorithm (excluding the preprocessing time), the total time for 500 iterations of Dijkstra's algorithm, and the ratio of execution time (time of Dijkstra's algorithm divided by time of the A* based algorithm). All the times are in seconds. All results have the objective value within 0.1% error.

The results shows that the execution times of Dijkstra's algorithm are 50-280% that of the A* search based algorithm. By comparing the results of `grid10` to `grid11`, `grid12` to `grid13`, and `grid14` to `grid15`, we see that the A* search based algorithm is faster than Dijkstra's algorithm when the number of demands are relatively smaller than the size of the network.

| Instance ID | Preprocessing time | A* search (a) | Dijkstra (b) | ratio (b/a) |
|---|---|---|---|---|
| `planar300` | 0.3 | 134.6 | 140.2 | 1.04 |
| `planar500` | 0.9 | 149.5 | 414.2 | 2.77 |
| `planar800` | 2.4 | 813.2 | 1093.2 | 1.34 |
| `planar1000` | 3.7 | 3202.2 | 1661.7 | 0.52 |
| `grid10` | 1.1 | 170.4 | 456.8 | 2.68 |
| `grid11` | 1.0 | 272.2 | 483.1 | 1.77 |
| `grid12` | 2.3 | 458.7 | 1028.0 | 2.24 |
| `grid13` | 2.3 | 954.8 | 1044.5 | 1.09 |
| `grid14` | 4.6 | 1323.4 | 2008.5 | 1.52 |
| `grid15` | 4.5 | 2538.0 | 2025.2 | 0.80 |

Table 2: Test results

## 5 Concluding Remarks

We show that the A* search based algorithm is faster than Dijkstra's algorithm when the number of demands is relatively smaller than the size of the network. The A* search based algorithm can be incorporated into the techniques [1] for reducing the number of iterations in solving the Lagrangian dual problem, which will yield acceleration for solving the multicommodity flow problem. We note that the A* search based algorithm can also be used for solving similar multicommodity network problems such as the multicommodity network design problem.

For a further study, improving the node potentials will accelerate the A* search. While we give the lower bounds of arc costs in this paper, we do not give the lower bounds of the lagrangian multipliers. If we can give the lower bounds of the optimal lagrangian multipliers, the node potentials will be further improved.

## References

[1] F. Banonneau, O. du Merle, and J.-P. Vial, "Solving large scale linear multicommodity flow problems with an active set strategy and Proximal-ACCPM," *Operations Research*, Vol. 54, No. 1, pp. 184–197, 2006.

[2] P. Chardaire and A. Lisser, "Simplex and interior point sprecialized algorithms for solving non-oriented multicommodity flow problems," *Operations Research*, Vol. 50, No. 2, pp. 260–276, 2002.

[3] E.W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, Vol. 1, pp. 269-271, 1959.

[4] A.V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," In *Proceedings of 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pp. 156–165, 2005.

[5] P.E. Hart, N.J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, pp. 100–107, 1968.

[6] T. Larsson and Di Yuan, "An augmented lagrangian algorithm for large scale multicommodity routing," *Computational Optimization and Applications*, Vol. 27, No. 2, pp. 187–215, 2004.