Annals of Operations Research 57(1995)73–101

Sequencing inputs to multi-commodity pipelines

Christopher A. Hane
CAPS Logistics, Inc., 2700 Cumberland Pkwy, Atlanta, GA 30339, USA

H. Donald Ratliff
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

This paper examines the problem of sequencing the input of commodities, e.g. petroleum products, to a pipeline so that a surrogate for pumping and maintenance costs is minimized. This problem is complicated by the need to impose a discrete framework which handles the sequencing choices on a continuous flow problem. By focusing on the discrete aspects of the problem, the proposed model allows decomposition of the sequencing problem into subproblems which can be easily priced out in a branch-and-bound algorithm. Computational results on data generated to mimic a large U.S. petroleum pipeline are presented. These results show that the branch-and-bound algorithm only explores a small region of the solution space within a reasonable amount of time, less than 2.5 minutes to optimally sequence deliveries to twenty-four destinations.

1. Introduction

The pipeline industry is unique among the transportation industries because it uses stationary carriers whose cargo moves rather than moving carriers of stationary cargo. This uniqueness leads to many benefits: no backhaul (deadheading), no packaging, and good reliability and safety.

Pipelines move approximately one-fourth of all inter-city freight in the nation.[1)] Traffic carried by pipeline companies in 1983 amounted to 67.1% of all railroad ton-miles, using only 6.4% of the labor of the railroad industry. This high level of automation implies the labor costs of pipelines are insignificant in comparison with the operating and fixed costs. Also, the automated nature of the pipeline industry allows for a higher level of central control than is possible with other carriers. This automation forces a tremendous amount of data processing at the control center.

[1)] All the statistics in the next three paragraphs were found in [7] with the exception of the information on Colonial Pipeline, which came from their own PR publication.

These control decisions have a major impact on the cost effectiveness of the entire operation. Indeed, at Colonial Pipeline Co., a large petroleum products pipeline with service from New Orleans, Louisiana to Newark, New Jersey, electric power costs alone represent 40% of all operating expenses.

The effect of the sequence of products in the line can be profound. For example, if a delivery of 100,000 barrels should be made from a batch of 150,000 barrels while the fluid is moving fairly quickly, then the pipeline may have to slow down so the delivery location can siphon out the required fluid. In the worst case, the flow of product may have to be stopped downstream of the delivery location to allow the delivery to be made. For petroleum products pipelines, these stoppages are extremely costly due to the viscosity and volume of fluid that must regain its lost momentum.

The main contribution of this paper is the formalization of an industrial problem. The groundwork laid by the model and algorithms should provide a fertile bed for further research into scheduling applications in the pipeline industry. While some of the assumptions found in this paper may impede its use in practice, the basic approach of decomposition and aggregation are worthwhile in trying to relax the assumptions which are restrictive.

This paper presents a sequencing algorithm to help the operator determine the cost implications of the sequence of products in the pipeline. The next section describes the sequencing problem and some of the issues which overlap our area of interest. Then we describe other optimization work in the pipeline industry. Section 3 introduces our discrete model of the sequencing problem together with our assumptions about how the pipeline operates. This section also introduces the basic definitions and some complexity results regarding our problem. The most interesting complexity result of this paper is that the simplest combination of pipeline structure and objective function possesses a lower bound which is arbitrarily poor, yet NP-hard to improve. Section 6 presents a greedy and a branch-and-bound algorithm for determining the sequence which minimizes a surrogate objective function of the sequence-dependent pumping and maintenance costs. The final two sections introduce the computational results and suggestions for further study. The computational work shows that the branch-and-bound algorithm explores a small fraction of the total number of solutions and the greedy algorithm quickly provides a reasonable solution. These preliminary results show the method and the problem area, in the least, deserve further study.

2. The problem

This work focuses on optimization of the transportation of various grades of refined and distilled petroleum products through a system of pipes. This fixed network is similar to that found in railroad and telecommunication application, but possesses significant differences. In telecommunications networks, some of the main concerns are route selection and network design (selecting arc capacities), and there is no control over the rate at which calls traverse an edge. Railroads are more similar to

pipelines than the telecommunications networks because the speed of the trains must be considered in the scheduling and only one train can occupy a track at any one time.

The major distinguishing factor between railroads and pipelines is the pipeline must always be full. For example, imagine a section of pipe where the direction of flow is uphill; if the volume of fluid in the pipe is less than the pipe volume, then there can be no flow out of the higher end of the pipe. This is the crucial aspect of flows in pipelines. It forces a *continuity* among the flows, which implies that local changes in the sequence of inputs have impacts on the sequence of deliveries that are far removed in terms of distance and time. This property is the heart of the difficulty of the problem which this paper addresses.

The pipeline system we sequence has many different commodities, a unique source, many delivery locations, no intermediate storage, and a simple path structure. This simplified pipeline system is a decomposition of a more complex system. Hence, the understanding and solution of the simple system is an important first step in the analysis of the complete system. The model is valid for multiple-source pipelines defined on acyclic graphs. Thus, the non-algorithmic results are more widely applicable than the algorithm developed. A description of the decomposition of a directed acyclic network of pipes into a collection of simple path systems is given in Hane [4].

The goal of the sequencing algorithm is to minimize the steady-state, sequence-dependent costs of delivering all the orders in a static order set. The true cost function is very complex, due to the different cost structures of the power companies that provide energy to the pipeline and to the maintenance costs of pumps. The energy costs are complicated by surcharges for the maximum usage that occurs during a particular time of day over the billing period. For example, if M_i = (maximum kWh used during peak time in day i), then the charge is $\$x$ per kWh + $\$y \cdot \max_i(M_i)$. Because of this apparent intractability of a true cost function, we will use the surrogate cost function described below.

A pipeline operates most efficiently when its pumps operate at their designed optimal performance level. Any deviation from this performance level results in a loss of pumping efficiency and, therefore, higher energy usage. Also, if the pumps are operating at a steady rate, there is less variance in the consumption of energy. This reduced variance makes the payment of any surcharge more palatable to management because all time periods have nearly equal energy consumption. Also, shutting down pumps to reduce energy consumption has an adverse effect on the maintenance costs because the mean time between repairs depends primarily on the number of shutdowns. Thus, one goal of the sequence should be to reduce the variance in energy demand, which may be achieved by limiting changes in pumping speed.

In discussing the sequencing problem with actual pipeline operators, one operator suggested that the clearest aspect of a sequence that distinguished its cost is the number of stoppages the sequence requires to deliver the orders. Thus, the more times a sequence must stop the flow in the pipeline to make a delivery, the more

costly it is. Since a stoppage can be very expensive (over $50,000 per event in a large diameter pipe), it makes sense to seek a sequence that minimizes the costs of all required stoppages. After the main pipeline is stopped to perform a delivery, the cost of that stoppage is related to how much work must be done to "restart" the flow and the maintenance cost incurred by the pumps.

Thus, we examine two cost functions which seek to minimize a weighted sum of the number of restarts required by the sequence. These ideas are made more rigorous in section 3 after we introduce the formal model.

2.1. SOLUTION METHODS

In pipeline systems, minimization of operating costs is often carried out by dynamic programming, nonlinear programming, or mixed integer programming methods [1,2,6]. The nonlinear models rely on solving systems of nonlinear fluid-flow equations to determine the relationships among inflows, pressures, pump speeds, and outflows. The system of equations is a snapshot view of the network. Therefore, it cannot model changes in operating conditions. These models excel at testing if all system parameters fall within operational guidelines.

However, the drawback to the nonlinear models is they fail to capture decisions made over time. To overcome this hurdle, the nonlinear model is solved as a subproblem in a dynamic programming framework. The dynamic programming algorithm is responsible for evaluating changes in the system state variables from one time period to the next. The drawback of this method is the time complexity of the algorithms. For each time period, the nonlinear fluid-flow problem must be solved. This methodology can be quite time consuming, and the level of detail may be unnecessary in the early planning stages. For example, in water distribution systems, Orr et al. [6] declare that the dynamic programming approach is limited to problems with less than two reservoirs. Joalland and Cohen [5] have developed a decomposition strategy to handle larger problem sizes. Their experiments are limited to three-reservoir problems.

The problem of sequencing the inflows of different commodities (usually various grades of refined or distilled petroleum products) has not received any attention in the literature. In practice, the inflows are sequenced by quality, lowest to highest then highest to lowest. This sequence minimizes the losses due to the mixing of adjacent commodities. It may be the case that mixed product can be sold as the lower grade (a mix of regular and high test gasoline sold as regular), or the product could require re-refining. However, this sequence does not tell the operator how to make deliveries, i.e. which part of the section of regular gasoline should be diverted to meet a demand.

The decision of where in a stream of product to remove product for delivery is an important one. The sooner the product is removed for delivery, the sooner there could be a pressure drop at the destination, which in turn could delay other downstream deliveries. This decision also determines the quality of the delivered product because the product mixes with adjacent products during transit.

3. The model

This section describes the model of the pipeline that is used in the algorithm. The model ignores the nonlinear fluid-flow equations and is combinatorial in nature.

A pipeline problem $\mathcal{P}$ is defined by the physical structure of the pipes and the static set of orders. The pipeline structure is modeled as a directed network $G(V, E)$, where V is the set of nodes and E the edges. Each edge E has an integral volume $\mathbf{v}(e)$, which represents the volume between the end nodes of the edge. We assume that G is a directed tree, i.e. the underlying undirected graph is acyclic. The nodes of V correspond to the sources, destinations, and junctions in the pipeline system. The set of orders O defines the commodity, input node, delivery node, and integral amount to be delivered for each order. The volume function $\mathbf{v}(\cdot)$ is defined on sets of edges or orders and returns the total volume of all the elements in the set. It will be clear from the context whether $\mathbf{v}(\cdot)$ is operating on edges or orders.

A *flow* in the pipeline is a connected set of edges which all contain fluid in motion at time t. An edge which is an element of a flow is *flowing*. If there is a single source or destination in the network, then there can only be one flow in any time period. Multiple source/destination pipelines may have many simultaneous flows. Without any loss of generality, we assume there is always at least one flow in every time period.

In order to realistically capture product mixing, a model must recognize fluid viscosities, turbulent flow, pipe drag coefficients, and a host of other data. These details make the sequencing problem too complex to solve in a reasonable amount of time. Also, this level of detail seems inappropriate for a strategic planning model. Thus, our model assumes no loss of products at the interface between orders. Thus, fluid in the pipeline can be classified as one of the commodities in the order set and does not change attributes in transit. If necessary, costs due to mixing can be captured in the objective function, but there is no means to handle product loss or migration of fluid from one commodity to another. This is discussed further in section 6.2.

We also assume each flow is a path. Thus, if any fluid moves on one edge out of a node, then no other edge out of that node can be flowing at that time. Similarly, if fluid moves into a node, then it can only do so on one edge at a time. This assumption forces flow to stop downstream of a delivery and upstream of an input. If there are multiple sources, there can be many node-disjoint flows during a single time period. This restriction is referred to as the path flow assumption.

Actual pipelines do not operate under this condition. Pipeline operators are capable of siphoning fluid out of the pipeline, or forcing fluid to enter, as the product moves through the line. The actual problem is further complicated by varying rates of flow along the line. The operators can slow down the flow in order to siphon off a specific product from the line as it moves by their station. The limits on how much product can be put in the line, how much removed, and what rates of flow are allowed depend on the operating characteristics of the pumps, valves, and pipes. These

characteristics are determined by the solutions to the fluid-flow equations for the system. A model which allowed variable rates of speed along the pipes, or splitting and merging of flows, must recognize the impact of these variables on the satisfiability of the fluid-flow equations. However, these are issues that we wish to avoid in the preliminary algorithm development.

So far, we have mentioned time periods, but have not formally defined them. In defining the time periods, there are three variables of interest: the unit of volume, the duration of a time period, and the rate of input. The model assumes that one unit of volume of product enters each flow during each time period. Now, fixing two of the three parameters above determines the third. For clarity, we choose the unit of volume for inputs as the same unit of measurement for the integral edge volumes. Hence, this unit of volume and the rate of flow determines the duration of a time period. In single-source pipelines, the constant rate-of-flow assumption is implied by the path flow assumption and conservation of volume (flow in must equal flow out in the time period).

Another basic feature of the model is that momentum propagates instantaneously through the pipe. Thus, if a quantity x of product enters at the source of a flow during the interval t to $t + h$, then a like quantity of material must exit that flow during the same interval. This follows directly from incompressibility of the fluids in the pipeline.

Lastly, we require that destinations are always able to accept delivery of its orders. There is no capacity or timing restriction on how much or when a delivery can be made, as long as the product that is removed from the pipeline at a node has that node as its destination in the order set.

To summarize, our model of pipeline flows makes the following assumptions:

- The pipeline is always full (the quantity of fluid in each edge must always equal the edge's volume) and product must be *pushed* out of the pipeline.
- One unit of fluid enters each flow during each time period.
- There is a constant rate of flow.
- There is no product loss at order interfaces.
- At any instant, the flows in the network must be a collection of node-disjoint simple paths.
- Momentum propagates instantaneously through the pipeline.
- Deliveries are unrestricted.

A helpful mental picture of this model is a set of railroad tracks (the pipes) filled with boxcars end-to-end. The edges of the pipeline network must be full at all times, so each edge e must contain $\mathbf{v}(e)$ boxcars. The nodes in V represent junctions, input or delivery locations. A flow in the pipeline at time t is one boxcar entering the network at a node (between two other boxcars, or after one boxcar if the node has degree 1) and pushing the boxcar in front of it, and it pushing the one in front of it, and so on, in a chain reaction down the track until one boxcar is pushed out of the network at its delivery node.

3.1. THE SCHEDULE

Before discussing the sequencing of inputs to a single-source pipeline, it will be beneficial to define a schedule of the pipeline operations. Then we can use the properties of a single-source pipeline to eliminate some of the timing and coordination issues in the more complex system and focus on the input sequence.

The schedule must delineate the activity in each of the time periods in the planning horizon. Thus, we define a schedule as the following set of data:

DEFINITION 1

A *schedule* indicates the source, destination, product entering, and product leaving the pipeline for each flow in each time period.

We also define an *input* schedule as the schedule restricted to only the source and entering product information. The *delivery* schedule is defined similarly on the delivery information.

A schedule for pipeline operations is difficult to develop for many reasons. First, the schedule must specify each flow during every time period. Initially, these flows are not known. Often one begins with an idea of the necessary inputs, and possibly some precedence constraints among the inputs, then determines the delivery schedule that could arise from the inputs. Since there is a one-to-one mapping from an input schedule to the delivery schedule, the input schedule alone is sufficient to determine the entire schedule [4]. However, the entire schedule is needed to evaluate the total costs.

When orders are not allowed to move off the unique source to destination path, it is always possible to have at least one flow per time period; thus, eventually all inputs can be made [4]. This follows directly from the assumption of unrestricted deliveries at the nodes and the acyclic topology of the network.

To make a schedule of flows on a directed acyclic graph, one can begin with precedence lists at each source. Thus, we begin with a partial ordering of the orders, where only orders that share a source are comparable. These lists can then be used to simulate the flows in the pipeline. (Since the pipeline must always be full, we need to know the product in the line, or linefill. Shortly, we will get rid of any dependence on this initial linefill and look at the steady-state system.) This simulation of flows may violate the path flow assumption because it only specifies sequences and not timing information. The easiest way to avoid a violation of the path flow assumption is to delay the inputs at the sources of all but one of the colliding flows. This delay has no effect on the precedence at any of the sources and prevents the collision. The simulation can continue in this manner, delaying inputs that cause a violation of the path flow assumption, until a schedule is built.

In choosing which inputs to delay to construct the schedule, a precedence among orders at each node in the network is defined. Thus, if unit volumes a, b and

c are waiting to traverse node v, specifying a precedence among these is equivalent to deciding which inputs must be delayed. If each node has a precedence list for the product that traverses it, then these lists will resolve the conflicts arising from colliding flows. In fact, since such a set of precedences must be non-contradictory, it is better to have a single precedence list for all nodes. This precedence list will determine the sequence that product can traverse the nodes of V. This precedence list of all products is called the *traversal sequence of* $\mathcal{P}$.

Certainly, requiring a traversal sequence to rank each unit of volume of the order set is extreme. Instead, the traversal sequence can be defined on orders, the elements of the order set O. If this is the case, then once an order begins to traverse node v, it blocks all other flow through v until it completes passage across v. This consequence of defining the precedence on orders captures a desire to not allow product mixture at junctions. Alternately, the traversal sequence could be defined on *batches*, where a batch is a contiguous section of product from a single order. When the pipeline operation only allows one batch per order, the pipeline is operating under *batch integrity* and the traversal sequence is *batch integral*. In section 6, it will be shown that batch integrity plays an important role in the complexity of the sequencing problem. Note also that for single-source pipelines, the input sequence and the traversal sequence of the source node are equivalent.

Using the boxcar model, an order $o \in O$ is a set of $\mathbf{v}(o)$ boxcars of one commodity travelling from s_o to d_o, $s_o, d_o \in V$. A batch integral traversal sequence dictates that these $\mathbf{v}(o)$ cars must be input to the network contiguously, but not necessarily in consecutive time periods, and cannot be separated during transit.

4. Feasibility results

This section considers the basic question: What does it mean for a pipeline problem $\mathcal{P}$ to be feasible? In investigating this question, we must also determine what orders are in the order set O. This leads to a definition of a steady state for the pipeline operation. From there, we develop the basic results which limit the types of schedules considered in the sequencing algorithm, much as basic job scheduling theory can be limited to permutation schedules.

For any pipeline problem $\mathcal{P}$, the last order input to the pipeline can *never* complete delivery in the time needed to input the order set because there is no way to push it away from its source. It seems that there are two resolutions to this problem. The first ignores the issue of delivering the orders in the order set and is only interested in inputting all the orders to the pipeline. The second alternative allows the order set to be extended. The first choice does not address the issue of feasibility in any meaningful way. Hence, we are left with deciding how to extend the order set.

Without resorting to stochastic arrivals, there are two ways to extend the order set. One is to assume the existence of an infinite amount of some filler product. This

filler can be used to push flow through the pipeline. The second extension repeatedly services the orders in the original set. The second extension corresponds to an assumption that future demands for service will closely mimic the current demands. If the pipeline is a petroleum product line, then sources may be production wells and destination refineries, or sources may be refineries and destination wholesalers. In either of these cases, it seems (highly) reasonable that service demands fluctuate about some norm. Therefore, the assumption of repeatedly servicing the original order set is reasonable.

Consequently, we will assume that given a pipeline problem $\mathcal{P}$, the order set O will be serviced ad infinitum. Since the orders are assumed to repeat, it makes sense that the input schedule must also be repetitive. Otherwise, clients may have to input product to the pipeline more often than their production rates allow.

DEFINITION 2

An input schedule is *reduced* to a sequence when the inputs are listed sequentially as they occur over time. All ties are broken according to a fixed ordering of the source nodes. The sequence determined by reducing the schedule is the *reduction* of the schedule.

DEFINITION 3

An input schedule is *fair* if its reduction is an infinite repetition of one permutation of the unit volumes of all the orders.

For example, let $\mathcal{P}$ be a pipeline problem defined on a path with one source and three demand nodes and two units of product going to each demand node. The reduction of the input schedule will be represented as a sequence of numbers, each digit i representing a unit of volume travelling to v_i. The sequence [112233, 112233,...] is fair because it is an infinite repetition of the ordered quantities; the sequence [121332, 121332,...] is also fair; [123213, 322113, 123213, 322113,...] is not fair because it repeats two different permutations of the quantities. Likewise, [111222333, 111222333,...] is not fair because it is not a repetition of a permutation of the ordered quantities.

We will shortly see that after a initial period the delivery schedule resulting from a fair input schedule will also repeat and the pipeline will be in a steady state. Fair input schedules allow many attributes to be defined on the time required to input one permutation of the order set, the *cycle length*. For single-source systems, the cycle length is also the volume of product in the order set. Since multiple-source systems can have more than one flow in each time period, there is no fixed relationship between cycle length and volume of the order set. In all pipelines, the set of flows during a cycle depends on the schedule. Thus, the schedule must be fixed before many questions about it can be answered.

We have assumed the schedule repeatedly services the same static order set. It then seems reasonable to also assume that the client wants his/her product to arrive at the destination at the same rate at which it is input to the pipeline. If the client has an order for 100,000 barrels per week, then the input amount should equal 100,000 barrels in every 7-day period. This is guaranteed by fairness. In order to force the delivered amount to equal the ordered amount in every cycle, we define the concept of a *balanced* schedule.

DEFINITION 4

Let $\mathbf{v}(O)$ be the total volume of the orders in O for $\mathcal{P}$, and π be its input schedule. We call π *balanced*, if after some finite start-up time, the volume of deliveries to each node during *every subsequence* of length $\mathbf{v}(O)$ in the reduction of the delivery schedule equals the ordered amounts.

This definition forces the reduction of the delivery schedule to be a repetition of a permutation of the unit volumes of the order set. The start-up time allows for a transitional period when the initial pipeline contents are delivered. This definition presupposes a result from Hane [4] that there is a one-to-one correspondence between input and delivery schedules, otherwise we could not define a property of the input schedule based on the delivery schedule.

Now we can present the two major feasibility results needed for the sequencing algorithm.

THEOREM 1

For any pipeline problem $\mathcal{P}$ defined on a directed acyclic graph, given a traversal sequence σ, a permutation of the unit volumes of the orders of O can be constructed which obeys σ and whose repetition is a fair schedule.

Proof

The detailed proof of this result can be found in Hane [4]. The underlying motivation for the result is that flow through a node v is ordered by σ and obeying that ordering only delays inputs at upstream locations. The tree structure of $\mathcal{P}$ prevents the flow upstream of v from interacting with downstream flow. □

This result shows that our assumption of fair schedules is not very restrictive, since such a schedule can be constructed from any traversal sequence.

THEOREM 2

In directed acyclic pipelines, only fair input schedules are balanced.

Proof

This proof can also be found in Hane [4]. Under a fair input schedule, an observer at a source sees exactly a permutation of that source's inputs enter the pipeline during a cycle length. Downstream of the source, an observer sees the input permutations of all upstream sources minus the deliveries made upstream. Therefore, the deliveries made upstream must also be permutations of the orders, and hence the schedule is balanced. □

This last result allows us to only consider fair input schedules in our scheduling algorithm. The assumptions we have made in this section (excluding the boxcar model of flow) are based on removing end effects that arise from a finite time horizon and finite order set. This leads to assuming the order set represents a mean level of requests for service to be met repeatedly. The repetition of the orders then leads straightforwardly to the requirements of fairness and balance. Together, fairness and balance require that the reduction of the input schedule be a permutation of the ordered amounts.

5. Backfilling

This section describes a procedure to determine the contents of a single-source path pipeline at the beginning of the steady state. The extension of these results to single-source pipelines defined on directed acyclic graphs is routine, but the extension to multiple sources is more difficult. It is important to be able to determine the linefill at the beginning of the steady state because we wish to avoid simulating the flows during the transitional period, and we must know the pipeline contents in order to determine the delivery sequence from the input sequence. Also, the procedure provides insight into the development of the sequencing algorithm.

Throughout the remainder of the paper, we assume that $\mathcal{P}$ is a single-source system defined on a simple path with nodes $v_0, v_1, \ldots, v_n$. This is a *simple* pipeline. The source is v_0 and the edges are $e_i = (v_{i-1}, v_i)$, $i = 1, \ldots, n$. The traversal sequence is a sequence of batches, $\pi = \{b_0, b_1, \ldots, b_m\}$, which obey a cyclic precedence $b_i \prec b_{i+1}$ and $b_n \prec b_1$ for traversing a node. The cyclic nature of the precedence arises from the need to make sure the flow across a $v \in V$ is fair, i.e. all of the ordered amounts which must cross v during an input cycle do so before product from the next input cycle is allowed to traverse v. There are three functions defined on the batches: $d(b_i)$ is the destination, $\mathrm{v}(b_i)$ the volume, and $p(b_i)$ the product type for the ith batch for $i = 0, \ldots, m$. If the pipeline operation is batch integral, then each batch is an entire order and $m + 1 = |O|$; otherwise, m is bounded from above by $\mathrm{v}(O)$. When there is one order to each node and the input sequence is batch integral, we will use the notation o_k to denote the order with destination v_k. Also, recall that for simple pipelines the traversal sequence and the input sequence are equivalent.

Let $t = 0$ be the time the first product from the order set O is input to the pipeline. Then the transitional period until a steady state is reached in the time from $t = 0$ until all the product which was in the pipeline at $t = 0$ has exited the pipeline. Once the last of the initial linefill has left the pipeline, it is easy to show that the pipeline contents are reproduced after every cycle of inputs [4].

The method of determining the pipeline contents at the beginning of the steady state is called *backfilling*. Two critical observations allow us to determine the pipeline contents at this time. First, the only batches that can travel on edge e_i are those whose destinations are v_j, $j \geq i$, i.e. the destination must be downstream of e_i. Second, if batches b_i, b_{i+1}, and b_{i+2} are consecutive in the traversal sequence, then the three batches must consecutively traverse any edge common to all their source–destination paths.

Product from the initial linefill will remain in the pipeline until a batch from π arrives at v_n. If we let t_0 denote the first such arrival time, then t_0 is the first time the pipeline fills with product entirely from the input schedule. Therefore, t_0 is the beginning of the steady state. Without loss of generality, we can assume that b_0 is a batch with destination v_n, because any batch preceding one to v_n will be pushed out of the pipeline before t_0.

At t_0, b_0 occupies that last $\mathbf{v}(b_0)$ units of the pipeline volume. The batch immediately upstream of b_0 must be the nearest batch, in terms of indices, following b_0 in the traversal sequence which has its destination in the last $\mathbf{v}(b_0)$ units of the pipeline (including the node $\mathbf{v}(b_0)$ units upstream of v_n if one exists). The procedure BACKFILL($\mathcal{P}$, π, n, k, i) returns the pipeline contents between v_n and v_k as a list l, where $l(j, m)$ is the index in π of the batch that occupies the mth unit of volume on e_j. This is an excessive representation of the contents, but it makes the notation much simpler.

The backfill algorithm is implemented using three pointers: one to the current edge being filled, one to the remaining volume on that edge, and one pointing into the traversal sequence at the last batch placed in the current edge, see figure 1. The algorithm loops over the traversal sequence and looks for the next batch which can be placed on the current edge. When this test succeeds, the remaining volume on the edge is decremented. If the batch has a volume greater than the volume remaining on the current edge, then the current edge pointer is decremented and the overflow volume is decremented from the new current edge. This continues until e_k is filled.

This algorithm is not polynomial because the amount of work to do in filling an edge depends on the edge volume and the excessive length of l. However, the algorithm does not need to compute the location of every batch in the pipeline. It only needs to compute the relative positions of the batches. Thus, edges with volume greater than $|\pi|$ can be replaced with edges that have volume bounded by $|\pi|$ by removing a volume equal to a multiple of the volume of product that traverses that edge in one input cycle (see Hane [4] for the full details). Therefore, each edge can be filled in one pass through the input sequence, and the original solution reported

```
BACKFILL( P, π, n, k, i)
    rv = v(e_n)
    while n > k do
        if ( d(b_i) ≥ n ) then
            if ( v(b_i) < rv ) then
                l(n, rv − v(b_i) + 1, ..., rv) = i
                rv = rv − v(b_i)
                i = i − 1
            else
                l(n, 0, ..., rv) = i
                n = n − 1
                rv = v(e_n) − v(b_i) + rv
                l(n, rv + 1, ..., v(e_n)) = i
                i = i − 1
            end if
            if ( i < 0 ) i = |π| − 1
        end if
    end while
    return l
```

Figure 1. The backfill algorithm.

by denoting, for each edge, in addition to l, the number of input cycles that were removed. For example, if an edge has volume 11, and the volume of product that must traverse it every input cycle is 4, then the edge volume can be reduced to 3 by removing 2 input cycles of volume. Hence, the 3 most downstream units of volume on the edge are followed by 2 repetitions of the 4 units of volume in the traversal sequence. The length of l is easily shortened by storing only the batch index and the location of the farthest downstream part of the batch. Thus, l has length bounded by $n \cdot |\pi|$ after removing the excess volume in an edge. Another aspect of the complexity of backfilling is the dependence on $|\pi|$. Without batch integrity, $|\pi|$ could be as large as $\mathrm{v}(O)$, otherwise $|\pi| = |O|$.

Forward-filling can be defined in an analogous manner by selecting an arbitrary order to occupy the initial position in the pipeline. However, forward-filling corresponds to an unknown point of time in relation to t_0.

Figure 2 shows the *flow matrix* F of a simple pipeline. Each unit of pipe volume is represented by a column, and the matrix element indicates its destination, i.e. F_{ij} represents the destination of the jth unit in the pipe at time i. Row i, $i = 0,\ldots,\mathrm{v}(O) - 1$, indicates the pipeline contents at t_i. The input sequence π is written to the left of the matrix; here, the number k represents a unit of volume travelling to v_k. This input sequence is batch integral (one batch per order) with $d(b_0) = v_6$ and $\mathrm{v}(b_0) = 2$. On the right is the delivery sequence. A bullet slightly offset above a column indicates the location of a destination. Deliveries are denoted by an up arrow (↑). Row 1 shows that one unit travelling to v_2 entered the pipeline and a delivery was made at v_3

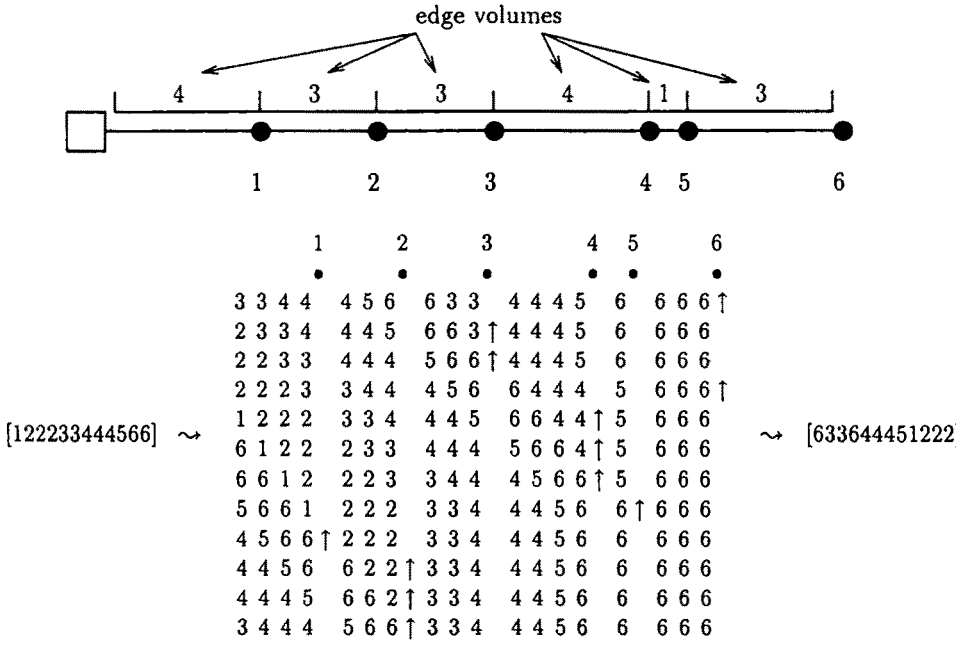

Figure 2. A pipeline and flow matrix.

during the time from t_0 to t_1. The number of rows in F is $\mathbf{v}(O)$, the total volume of the orders. The number of columns in F is $\mathbf{v}(E)$, the volume of the pipeline.

The first row of the flow matrix is the result of backfilling the pipeline. The pipeline fills backward from v_6 toward the source according to the traversal sequence. The order to v_6 is the only order that can occupy e_6. Since e_6 has volume 3, two orders to v_6 must be placed consecutively in e_6. The traversal sequence determined that o_5 pushes o_6. o_5 is pushed by o_4, which in turn is pushed by o_3. After placing o_3 in the line, there is one more unoccupied unit of volume on e_3. According to the traversal sequence, o_2 pushes o_3; however, placing o_2 here would put it downstream of its destination. Similarly, o_1 cannot be placed here. Thus, o_6 must be the order in this position in the pipeline.

To determine the relation between row i and $i+1$, we use the location vector $L(l)$, $l = i,\ldots,\mathbf{v}(E)$:

$$L(l) = \begin{cases} k & \text{if } l \text{ is the volume separating } v_0 \text{ from } v_k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $L(k) = 0$ if there is not a delivery node k volume units from the source, otherwise k is the distance (in volume) from the source to $v_{L(k)}$, e.g. in figure 2, $L(10) = 3$ because v_3 is 10 units away from the source. Row $(i+1)$ is derived from row i by letting j^* be the minimum index such that $F_{ij} = L(j)$, then

$$F_{(i+1)(j+1)} = \begin{cases} F_{ij} & \text{for } j = 1, \ldots, (j^* - 1), \\ F_{j(j+1)} & \text{for } j = j^*, \ldots, m, \\ \text{next input} & \text{for } j = 0. \end{cases}$$

6. The sequencing algorithm

In our model, the path flow assumption forces flow to stop downstream of a delivery and upstream of an input. Thus, our model simplifies the real-world problem by assuming that every input and delivery causes a stoppage instead of a slowdown. The cost incurred by the pipeline is not in the stoppage of the flow but in the restart of the flow.

In this and the following sections, the product type plays no role in the sequencing algorithm. The product type plays a role in the sequencing decisions to the extent that mixing costs are significant. We will see that if we totally ignore all mixing costs (or there is a single commodity), the sequencing decision is trivial. On the other hand, if all orders are distinct products, then batch integrity captures some of the mixing restrictions, and others can be captured in the objective function. In describing the algorithm, it will become clear that having orders share a product type is actually a relaxation of the batch integrity rule, and thus makes the sequencing problem easier. So we will use batch integrity to limit mixing and explain how to incorporate other mixing costs in the objective after describing the algorithm.

Our objective function models the cost of the restarts in a schedule. After a delivery at v_i at time t, there is a *restart* at time $t+1$ if the delivery at time $t+1$ occurs at v_j, $j > i$. The cost incurred at time $t+1$ is either the number of edges (the cardinality case) or a weighted sum of the volumes of edges that were not in a flow at time t, but in a flow at time $t+1$. If the delivery at time $t+1$ is upstream of v_i, there is no restart cost in that time period. The sequencing algorithm seeks to minimize the cost of restarts in an input cycle. We call this the restart problem for $\mathcal{P}$.

Now, let us ask the question: What makes an input sequence have a high restart cost? In general, we wish that the next downstream delivery following delivery at v_i occurs close to v_i. Only the downstream deliveries are of interest because an upstream delivery does not cause a restart. If the total restart cost of a sequence could be limited to one restart per edge, then that sequence must be optimal because that value is clearly a lower bound on the cost. Ensuring that the next downstream delivery is as close as possible to v_i is a myopic decision that is used in the greedy algorithm.

Now, we wish to establish how good the lower bound of $n-1$ restarts can be for minimizing the number of edges restarted during an input cycle. The worst cardinality restart cost any sequence can achieve is to have a delivery at v_n follow every other delivery. Thus, if K is a bound on the volume of orders to the first $n-1$

delivery nodes and the volume to deliver at v_n exceeds $K(n-1)$, there could be a sequence that restarts from v_i to v_n K times, requiring a total of $K/2 \cdot n(n-1)$ total edge restarts. Alas, there is a class of simple pipeline problems that require $n/2 \cdot (n-1)$ restarts for all input sequences [4]. In these problems, all the orders have unit volumes except o_n, which has volume $n/2 \cdot (n+1)$ and $\mathbf{v}(e_i) = n - i + 1$. Because the number of restarts for this class grows with n^2, the lower bound of $n-1$ can be arbitrarily far from the optimal. This class of problems establishes the following result.

THEOREM 3

No heuristic H can solve the batch integral, cardinality restart problem on a simple pipeline with worst-case performance bounded by $K(n-1)$ for any fixed K, where n is the number of delivery nodes.

Proof

There exists a class of problems whose optimal number of cardinality restarts is $n/2 \cdot (n-1)$. Therefore, choosing $n > 2K$ yields a counter-example to any such H. □

Suppose we have a 3 delivery node pipeline to sequence. To restart e_3 only once, the sequence must force all deliveries at v_2 immediately after or before all deliveries at v_1. Otherwise, e_3 must restart after a delivery to v_1 because the next delivery is not at v_2, and restart again after delivery at v_2 because the next delivery is not at v_1. To have the deliveries at v_2 immediately follow those at v_1, there must be a subset of orders with volume equal to $\mathbf{v}(e_2) - \mathbf{v}(v_2)$, where $\mathbf{v}(v_2)$ is the volume of all orders to v_2. If the sequence must be batch integral, there is a reduction from the partition problem to this consecutive time period delivery problem. The elements a_i in the partition problem are the orders going to node v_3 with volumes equal to their size. The pipeline has three delivery nodes, and unit edge volumes except $\mathbf{v}(e_2) = (\sum_i a_i)/2 + 1$. The orders to nodes v_1 and v_2 are unit volumes. The only way the delivery at v_2 can immediately follow that at v_1 is if o_2 is located adjacent to v_2 when o_1 is adjacent to v_1. If this is the case, then the orders that separate o_1 from o_2 corresponds to a solution to the partition problem. To complete a proof of NP-completeness, the membership of the problem in NP must be shown. The polynomial verification of a candidate solution is performed by backfilling e_2 beginning with o_2 adjacent to v_2. This establishes the following results.

THEOREM 4

The simple consecutive delivery problem: Given a simple pipeline $\mathcal{P}$, is there a batch integral input sequence which forces delivery at v_2 immediately after v_1? is NP-complete.

COROLLARY 1

The consecutive delivery problem: Given a pipeline problem $\mathcal{P}$ and two distinguished delivery nodes v and ρ, is there a batch integral input sequence which forces delivery at v immediately after ρ? is NP-complete.

Proof

By restriction to the case of theorem 4. □

Without batch integrity, this problem is easily solved by splitting whatever batch overlays v_1 when o_2 is ready for delivery and inserting o_1. Thus, batch integrity is the crucial property in determining the complexity of the sequencing problem.

Suppose we had a polynomial procedure to optimally solve our restart problem and applied it to our transformation of a partition problem. The only possible number of edge restarts is 2 or 3, depending on whether e_3 is restarted twice. So any procedure which can find the optimal solution to this restart problem can show there is no solution to the partition problem.

THEOREM 5

The batch integral, cardinality restart problem on a simple path pipeline is NP-hard.

Proof

Since partition is NP-complete, co-partition is NP-hard [3]. We have shown a reduction from co-partition to finding the optimal solution to this restart problem. □

More interestingly, the fact that there are only two possible objective values to the three-node restart problem gives rise to the following results.

COROLLARY 2

Improving the lower bound of $(n-1)$ edge restarts, for the batch integral, cardinality restart problem on a simple path pipeline is NP-hard.

Proof

Any algorithm which can improve the lower bound of two restarts for our transformation of a partition problem can solve the co-partition problem. □

COROLLARY 3

No algorithm can solve the batch integral, cardinality restart problem on a simple path pipeline in polynomial time with a worst-case performance ratio of less than 1.5 unless P = NP.

Proof

Any such algorithm would be able to solve the partition problem using the transformation to the above three-node pipeline problem. □

If there is any hope amidst these negative results, then it is in the fact that the reductions are from partition, a problem for which there is a pseudo-polynomial dynamic programming algorithm [3].

6.1. GREEDY SEQUENCING

This section develops a greedy algorithm (Greedy Restart Algorithm, GRA) for the restart problem defined on a simple pipeline. This greedy algorithm is constructive and builds the sequence by adding batches to a partial sequence of batches. Again, we are dealing only with a simple pipeline (single-source and simple path structure). The algorithm has the flexibility to force batch integral sequences, or follow other product mixing rules.

Suppose the pipeline has two destinations; then any sequence which inputs the batches to v_1 consecutively is optimal because it restarts the last edge only once per cycle. If another destination is added to the pipeline, we can insert orders to the new destination into the current input sequence. Performing this insertion at minimum cost is the greedy decision. The important feature of the algorithm is to begin with nodes v_n and v_{n-1} and add the nodes closer to the source, instead of the seemingly more natural approach of adding nodes to the end of the partial pipeline.

If an order to be delivered at v_1 at time t is added to a current sequence, this new delivery will interrupt the delivery currently scheduled at t, which will resume after this new delivery is completed. Therefore, given the delivery and input sequences, the marginal cost of inserting an order to the first delivery node is easy to compute.

The GRA seeks to insert a batch to v_j to the partial input sequence so that is causes a restart to v_{j+1}; if this is not possible, it checks for a restart to v_{j+2}, and so on. When adding node v_{n-k} to the partial pipeline, the algorithm possesses an input and delivery schedule for all orders to the nodes from v_{n-k+1} to v_n. See figure 3 for the outline of the algorithm using the cardinality objective function and batch integrality. Beginning with $m = 1$, the partial input sequence is used to repeatedly backfill the pipeline from v_{n-k+m} to v_{n-k} for increasing values of m. For a fixed m, the backfilling begins by placing a batch to v_{n-k+m} at that node. The pipeline contents after backfilling determine if a batch interface (the place where two batches meet) traverses v_{n-k} while a delivery is made at v_{n-k+m}. If a traversal occurs, then all batches to v_{n-k} are inserted into the sequence between the batches that define the batch interface. Since this procedure is performed sequentially for increasing values of m, the batches to v_{n-k} are inserted for minimum cost. Note, to be sure the batches are inserted for minimum cost, the algorithm must backfill from v_{n-k+m} for each batch with that node as its destination.

```
Begin
k = 2, π = { all orders to ν_n followed by orders to ν_{n-1} }.
while n − k > 0 do
    for m = 1 to k do
        for each b_i ∈ π with d(b_i) = n − k + m do
            l = BACKFILL( P, π, n − k + m, n − k, i)
            j = l(n − k + 1, 1), j is index of batch incident to ν_{n-k} on e_{n-k+1}
            if the volume of b_j upstream of ν_{n-k} is less than v(b_i) then
                insert all batches to ν_{n-k} between b_j and b_{j+1} in π
                cost = cost + m
                m = k + 1; k = k + 1; break;
            end if
        end for b_i
    end for m
end while
return π, cost
```

Figure 3. The Greedy Restart Algorithm.

This procedure is amenable to many definitions of a batch interface. Indeed, the definition of a batch interface determines the policy for allowing product mixing. Under batch integrity, the batches are orders and batch interfaces are the locations where different orders meet. Alternately, the GRA could recognize the product type of a batch and allow a similar product to split the batch in transit, as discussed below.

6.2. MIXING COSTS

The issue of the effect of the product type on the sequencing problem has not yet been addressed. If all orders are the same commodity, then there are no mixing costs. The pipeline contains only one type of product, so all delivery nodes are ready to accept delivery at all times. Therefore, the operator can select the delivery sequence that removes the order quantities sequentially from ν_1 to ν_n. This sequence restarts each edge only once and must be optimal.

Batch integrity can be viewed as every order defining its own commodity. Then to minimize the amount (not the cost) of mixing, the operator is forbidden from splitting an order into many batches, or in the multiple-source case, from inputting a product to the pipeline when it would split an order in transit.

Most mixing that occurs in the pipeline is due to the displacement of a lighter viscosity fluid by a heavier one as the heavy fluid settles to the bottom of the pipe. To combat the settling of the heavier fluid, the products are transported in a turbulent flow. While this increases the mixing locally, it limits the settling of the heavier fluid, thus decreasing the total amount of mixed product. In spite of the turbulent flow, the area of mixed product can extend for up to two miles in a 36" diameter pipe.

As stated previously, the cost of product mixing can be captured in the objective function of the restart problem. When b_i is inserted into the sequence between b_j and b_{j+1}, the mixing cost is computed as the cost due to having b_i travel adjacent to these two batches from v_0 to $v_{d(b_i)}$ minus a correction term removing the mixing cost for having b_j and b_{j+1} adjacent for the same distance. By pricing out the insertions in this way, the algorithm can remove the constraint of batch integrity and handle mixing costs as a function of the distance two batches travel while adjacent.

6.3. BRANCH-AND-BOUND ALGORITHM

The GRA described in the previous section is easily adapted to a branch-and-bound, B&B, framework. The algorithm uses the same basic design as the GRA, except all possible insertions to the current partial sequence are priced out. The backfill procedure allows the evaluation of each subproblem.

Our implementation of the B&B algorithm first used the greedy algorithm to obtain a good feasible solution to prune the tree. The implementation then used a best-bound search to choose the next subproblem to evaluate, see figure 4. The best bound rule is enforced by recursively calling the next subproblem immediately after not fathoming the current solution.

```
Subproblem(k, π, bestcost)
  while n − k > 0 do
    for m = 1 to n − k do
      for each b_i ∈ π with d(b_i) = n − k + m do
        l = BACKFILL(P, π, n − k + m, n − k, i)
        j = l(n − k + 1, 1), (j is index of batch incident to ν_{n−k} on e_{n−k+1})
        if the volume of b_j upstream of ν_{n−k} is less than v(b_i) then
          cost = cost + m
          if ( cost + n − k − 1 > bestcost ) return (fathomed)
          insert all batches to ν_{n−k} between b_j and b_{j+1} in π
          m = m + 1
          Subproblem(k + 1, π, bestcost)
        end if
      end for b_i
    end for m
    if ( cost < bestcost and n − k = 1 )
      bestcost = cost
      π* = π
    end if
  end while
  return π*, bestcost
```

Figure 4. The branch-and-bound algorithm.

The objective value of the partial input sequence is bounded by adding to its restart cost the cost of restarting the remaining edges of the pipeline just once. If this bound on the objective for the partial sequence exceeds the current best solution, then this partial sequence cannot be optimal and is discarded.

7. Computational results

In order to test the greedy and B&B algorithms, a number of random problem sets were created and analyzed. The problems in each data set are presented in increasing order of lower bound. There was just one order per destination in all the data sets generated. If multiple orders to a destination are allowed, then the problems are equivalent (since there are no mixing costs) to allowing certain orders to be split. The solutions to these problems should be better than the same problem solved without splitting.

The ranges for the data were selected to approximate a main line of Colonial Pipeline. Colonial Pipeline serves the eastern U.S. with two main lines that vary from 40" to 32" in diameter, with minimum order quantity of 75,000 barrels. The length of both main lines is 2885 miles, with approximately twenty junctions and delivery facilities on each. Assuming a pipe diameter of one meter to simplify the calculations, a minimum order is 421,000 ft^3, or 9.44 miles long.

All data were selected from a uniform distribution. In Data Set 1, the number of destinations varied from 5 to 30, the edge volumes from 1 to 20, and the order volumes from 1 to 10. Table 1 describes the first set of ten problems. The lower

Table 1

Data Set 1.

Problem number	Ordered volume	Pipeline length	Lower bound	Number of destinations
0	42	64	47	7
1	23	60	47	6
2	49	76	67	8
3	49	108	100	8
4	53	122	103	10
5	81	148	146	17
6	107	175	159	19
7	77	182	165	17
8	108	226	223	23
9	126	259	243	24

bound is the lower bound for the weighted restart problem, i.e. the length of the pipeline from the first destination to the end of the pipeline. The lower bound for

the cardinality restart problem is the number of destinations minus one. The problems generated were solved twice; once with the cardinality objective function that assumes all edges have the same restart cost, and then with a weighted objective that uses the volume of the edge as the restart cost of the edge. All reported execution times are on a Sun SparcStation1.

In comparison to the Colonial main line, the larger examples have comparable line length and number of destinations. The range of order volumes is slightly smaller than the true range; however, the greedy algorithm is always optimal if the order volumes exceed the edge volumes [4].

Table 2 shows the outcome of the cardinality GRA applied to the problems in Data Set 1. The column labelled "# of B&B nodes" is the number of subproblems examined by the B&B procedure. The number of sequences B&B may have to explore is $(n-1)!$, where n is the number of delivery nodes in the problem. A "0"

Table 2

Cardinality greedy backfill solutions. Data Set 1.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	7	7	6	1.00	26	0.194
1	5	5	5	1.00	0	0.054
2	8	8	7	1.00	15	0.207
3	7	7	7	1.00	0	0.160
4	10	10	9	1.00	303	1.534
5	18	17	16	1.06	2655	13.641
6	20	20	18	1.00	228	2.627
7	21	20	16	1.05	1028	4.749
8	24	24	22	1.00	8196	67.784
9	27	23	23	1.17	9659	90.888

in the column for the number of B&B nodes indicates that the heuristic solution achieved the lower bound and no branching was needed. The reported CPU seconds is the total time to find the greedy solution and perform the B&B optimization.

Seven of the ten problems in this set were solved optimally by the GRA. The computational effort for the B&B depends on the number of nodes in the pipeline as well as the size of the search tree. The time spent in performing the GRA depends on the nodes in the pipeline and how many insertions must be attempted before the minimum cost insertion is found. For all of these problems, the GRA took less than 2 seconds to solve.

The weighted problems, table 3, are somewhat more difficult to solve than the cardinality ones. This is expected because the GRA solution to both the weighted and

Table 3

Weighted greedy backfill solutions. Data Set 1.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	53	53	47	1.00	31	0.213
1	47	47	47	1.00	0	0.055
2	81	76	67	1.07	42	0.301
3	100	100	100	1.00	0	0.162
4	119	104	103	1.14	948	4.728
5	171	150	146	1.14	7664	40.783
6	182	182	159	1.00	4231	32.514
7	210	202	165	1.04	1358	6.871
8	250	231	223	1.08	16876	136.468
9	266	243	243	1.09	15418	145.533

Table 4

Problems for Data Set 2.

Problem number	Ordered volume	Pipeline length	Lower bound	Number of nodes
0	40	47	43	8
1	40	51	46	8
2	41	72	64	8
3	56	79	67	10
4	44	81	73	9
5	34	76	75	8
6	51	86	79	9
7	59	82	79	12
8	50	103	85	9
9	60	94	92	12
10	72	114	99	11
11	36	113	109	9
12	54	132	113	10
13	51	121	114	10
14	62	128	120	10
15	70	136	124	12
16	69	141	134	12
17	67	144	130	12
18	56	145	139	12
19	61	163	153	12

Table 5

Cardinality greedy backfill solutions. Data Set 2.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	8	8	7	1.00	28	0.189
1	7	7	7	1.00	0	0.112
2	7	7	7	1.00	0	0.097
3	10	9	9	1.11	50	0.371
4	8	8	8	1.00	0	0.195
5	8	8	7	1.00	18	0.147
6	10	9	8	1.11	80	0.438
7	15	15	11	1.00	1124	3.857
8	8	8	8	1.00	0	0.159
9	15	14	11	1.07	228	1.170
10	12	12	10	1.00	36	0.435
11	13	10	8	1.30	327	1.341
12	12	9	9	1.33	82	0.643
13	10	9	9	1.11	62	0.581
14	10	9	9	1.11	63	0.456
15	13	13	11	1.00	492	2.515
16	13	11	11	1.18	177	1.298
17	11	11	11	1.00	0	0.418
18	11	11	11	1.00	0	0.330
19	14	13	11	1.08	78	0.743

Table 6

Weighted greedy backfill solutions. Data Set 2.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	51	44	43	1.16	161	0.570
1	46	46	46	1.00	0	0.099
2	64	64	64	1.00	0	0.114
3	77	67	67	1.15	131	0.658
4	73	73	73	1.00	0	0.159
5	76	76	75	1.00	18	0.149
6	100	82	79	1.22	197	0.894
7	103	103	79	1.00	3891	13.226
8	85	85	85	1.00	0	0.198
9	126	113	92	1.11	478	2.139
10	120	107	99	1.12	128	0.925
11	190	143	109	1.33	424	1.702
12	153	113	113	1.34	82	0.654
13	127	114	114	1.11	161	0.875
14	136	120	120	1.13	118	0.890
15	148	148	124	1.00	1231	5.836
16	160	134	134	1.19	237	1.628
17	130	130	130	1.00	0	0.423
18	139	139	139	1.00	0	0.326
19	193	185	153	1.04	78	0.756

cardinality problems are the same. In spite of this deficiency, the worst solutions are only 14% more than the optimal answer.

The problems of Data Set 2, table 4, were generated like those of Data Set 1, except the number of nodes is U[8,12]. The results for this data set are also good. In the cardinality case, eleven of the twenty problems were solved optimally by the heuristic, and only two sequences had values more than 20% above the optimal. The weighted case showed similar behavior to that of the first data set, with nine optimal heuristic solutions and two greater than 20% above the optimal.

The next set of problems were generated differently. The GRA is optimal if the volume of each order exceeds the volume of the edge before its destination. Therefore, we created a data set which had harder problems by doubling the size of all orders going to the second half of the pipeline and doubling the volume of the first half of the edges. This transformation of the data tries to force expensive restarts for the deliveries near the source. These problems are presented in table 7. Prior to the transformation, the edge volumes are U[5,30], the order volumes are U[1,10], and the number of nodes U[8,12].

Table 7

Problems for correlated Data Set 3.

Problem number	Ordered volume	Pipeline length	Lower bound	Number of nodes
0	130	144	143	10
1	83	146	145	8
2	78	161	160	9
3	97	162	161	9
4	88	163	162	8
5	108	173	172	9
6	113	183	182	10
7	72	192	191	8
8	132	195	194	10
9	87	200	199	9
10	117	212	211	10
11	97	221	220	10
12	55	228	227	10
13	70	229	228	8
14	68	230	229	9
15	78	231	230	10
16	92	236	235	10
17	109	254	253	11
18	82	275	274	11
19	105	276	275	11

The motivation for this set of problems was that they would be more difficult than the previous data sets. In comparison to Data Set 2, where eleven problems had

Table 8

Cardinality greedy backfill solutions. Correlated Data Set 3.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	12	12	9	1.00	225	2.023
1	9	9	7	1.00	127	0.918
2	10	10	8	1.00	121	0.926
3	13	11	8	1.18	340	2.325
4	12	8	7	1.50	147	1.117
5	10	10	8	1.00	41	0.650
6	12	11	9	1.09	192	1.762
7	7	7	7	1.00	0	0.288
8	13	13	9	1.00	342	2.975
9	9	9	8	1.00	19	0.483
10	14	13	9	1.08	627	5.680
11	11	11	9	1.00	67	0.853
12	10	9	9	1.11	52	0.566
13	10	8	7	1.25	40	0.527
14	17	12	8	1.42	888	5.745
15	12	12	9	1.00	154	1.200
16	9	9	9	1.00	0	0.491
17	12	12	10	1.00	146	1.471
18	13	11	10	1.18	279	2.372
19	19	15	10	1.27	2714	20.880

Table 9

Weighted greedy backfill solutions. Correlated Data Set 3.

Problem	Greedy cost	Optimal cost	Lower bound	Greedy/Opt ratio	# of B&B nodes	CPU seconds
0	211	191	143	1.10	609	4.825
1	182	167	145	1.09	153	1.095
2	180	180	160	1.00	150	1.129
3	292	206	161	1.42	847	5.435
4	312	169	162	1.85	147	1.151
5	197	197	172	1.00	41	0.660
6	238	203	182	1.17	158	1.622
7	191	191	191	1.00	0	0.289
8	262	255	194	1.03	527	4.497
9	215	215	199	1.00	23	0.502
10	351	285	211	1.23	1003	7.854
11	268	250	220	1.07	102	1.120
12	281	227	227	1.24	195	1.449
13	281	242	228	1.16	40	0.331
14	482	311	229	1.55	1345	8.839
15	283	283	230	1.00	208	1.456
16	235	235	235	1.00	0	0.497
17	324	299	253	1.08	542	4.560
18	355	297	274	1.20	990	8.117
19	484	382	275	1.27	5433	43.836

weighted optimal solutions equal to the lower bound, in Data Set 3 only two did. In spite of this, eleven cardinality problems were solved optimally by the heuristic, with four solution values more than 20% greater than the optimal value. The worst cardinality solution reported, with value 1.5 times the optimal, is a member of this data set. The weighted case had only six optimal heuristic solutions, with seven solution values 20% more than the optimal value. The average number of nodes searched for the weighted problems in Data Set 3 is almost twice that of Data Set 2. This indicates that the lower bound is not as helpful in pruning potential solution from the search tree. This can be verified by comparing the values of the optima to the lower bound in table 9.

As problem size increases, the computation required for the B&B algorithm increases exponentially. This can be seen already in the problems of Data Set 1, which vary in size from 6 to 24 delivery nodes with run times from 0.2 to 145 seconds. Consequently, the need for a good heuristic increases with problem size, even if only to provide a good candidate solution to promote fathoming. The greedy method adequately serves this purpose, granting a reasonable quality solution quickly. The development of the heuristic also helped the B&B implementation, by allowing fast augmentations of the partial solutions of each subproblem.

Determining an estimate of the maximum problem size one could solve with these techniques is difficult because the performance of the methods is so data dependent. However, in all the problems the evaluation of a subproblem required less than 0.01 seconds, so a large number of sequences can be evaluated quickly. To test how large a problem could be reasonably solved, five random cardinality problems with 50 delivery nodes were generated and solved. Of these five problems, only one found a provably optimal solution in less than 500,000 nodes. (All problems were terminated after evaluating 500,000 nodes.) The maximum time spent in the GRA was 6.77 seconds, and the maximum time spent in B&B was 3970.54 seconds on an IBM RS 6000 model 550 (which is roughly four times faster than the computer used for the other runs). This performance degradation can be mainly attributed to the lack of a sophisticated branching rule; branching is always performed as depth-first search. The GRA's worst performance was 24% above the lower bound.

8. Further work

The model of pipeline flows presented in this paper is quite simple, yet the problems defined with it are difficult. The simplicity of the model is helpful in introducing the optimization aspects of the sequencing problem to pipeline operators untrained in OR techniques. The principal insight of the backfill procedure, that one can build the pipeline backwards from a pure sink to a pure source node, is useful in decomposing more complex systems. A next step in this work is the generalization of the sequencing algorithm to a scheduling algorithm for multiple-source pipelines defined on directed acyclic graphs.

Multiple-source pipelines raise numerous issues that have been avoided in the earlier discussions. Some of these issues arise from the flexibility of allowing more than one flow in a time period. A significant difference between the operation of single-source versus multi-source pipelines is the allowance of idle time. For example, if flow is not allowed through a node during a set of time periods, the network decomposes into components that can have independent flows. If one component has less product to move internally than another component, when should it be idled, and after idling when and how should it be restarted so that it gradually builds up a set of flowing edges to reach the other components?

Another difficult issue in scheduling a multi-source pipeline is the routing of product. Even though the pipeline is acyclic, the operator can use the input of a commodity at one location to provide product for any delivery of the same commodity.

There are a number of other issues that must be addressed in order to provide a practical decision aid for the pipeline operators. The model and algorithm must handle the pumping costs more realistically by incorporating some notion of the friction that determines the loss of energy as the product moves through the pipe. This extension would not require much modification of the current model, but the subproblems to be evaluated would become much more difficult.

The most important extension of the model would be in allowing flow to split or join at nodes in the network. This extension seems to open a host of difficult issues related to the feasibility of the desired flows, i.e. given a set of desired flows in the network, is there a set of control variables (pump speeds and valve positions) that induce the desired flows? Thus, a successful system will incorporate these optimal control issues into the discrete framework for making the sequencing decisions.

Hopefully, this work will spark interest in a new area of scheduling applications and generate new ideas and techniques for cyclic scheduling problems in other areas as well.

Acknowledgements

The authors wish to thank David Goldsman for his support and helpful comments during the investigation of this topic. This work was supported by the Office of Naval Research, Contract Number N00014-89-J-1571.

References

[1] B. Coulbeck and C.H. Orr, An applications review of modelling and control of water supply and distribution systems, in: *Computer Applications in Water Supply*, Vol. 2, ed. B. Coulbeck and C.H. Orr (Research Studies Press Ltd., Letchworth, Hertfordshire, England, 1988) pp. 165–186.

[2] B. Coulbeck, A review of methodologies for modelling and control of water supply, in: *Computer Applications in Water Supply*, Vol. 2, ed. B. Coulbeck and C.H. Orr (Research Studies Press Ltd., Letchworth, Hertfordshire, England, 1988) pp. 80–109.

[3] M.R. Garey and D. Johnson, *Computers and Intractability* (Freeman, New York, 1979).

[4] C.A. Hane, Scheduling multi-product flows in pipelines, Ph.D. Thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology (1991).
[5] G. Joalland and G. Cohen, Optimal control of a water distribution network by two multilevel methods, Automatica 16(1980)83–88.
[6] C.H. Orr, B. Coulbeck, M. Brdys and M.A. Parkar, Computer control of water supply and distribution systems: Structures, algorithms and management, in: *Computer Applications in Water Supply*, Vol. 2, ed. B. Coulbeck and C.H. Orr (Research Studies Press Ltd., Letchworth, Hertfordshire, England, 1988) pp. 392–420.
[7] F.J. Stephenson, Jr., *Transportation U.S.A.* (Addison–Wesley, 1987).