

Scheduling of Multi-Product Fungible Liquid Pipelines using Genetic Algorithms

D. Scott Crane
Williams Energy Group
Tulsa, OK, USA
Cranes@weg.twc.edu

Roger L. Wainwright
Math and Computer Sciences
The University of Tulsa
Tulsa, OK USA
Rogerw@utulsa.edu

Dale A. Schoenefeld
Math and Computer Sciences
The University of Tulsa
Tulsa, OK USA
Schoend@utulsa.edu

Keywords: Pipeling Scheduling, Combinatorial Optimization, Genetic Algorithms

Abstract: This paper explores the application of genetic algorithms (GA) to the problem of scheduling product movement in a multi-product, fungible, liquids pipeline. The storage of, and demand for product at multiple terminal facilities are modeled as well as the one way transfer of products between the pipeline terminals. The GA is driven to its optimal or near-optimal solution by a table-based set of product volume goals for each terminal and by penalties for non-feasible solutions that do not represent a valid pipeline schedule. Our model was tested with great success. The significance of this research is (1) this problem, to the authors' knowledge, has not been solved before using evolutionary computation methods or any other method other than brute force or by applying previous experience, and (2) our GA model for constructing the chromosome and applying rules as we step through the pipeline can (in theory) be scaled up for a more complicated (industrial size) pipeline network which includes more products, terminals, edges and links.

1. INTRODUCTION

Pipeline scheduling is a difficult optimization problem. Liquid pipeline scheduling is unique in that the products in the pipe literally push each other along. In addition to the product stored in terminal tankage, pipelines must always contain enough product to completely fill the pipeline. It is impossible to send a product from terminal to terminal without displacing the product already in the line (called linefill). It is the goal of the pipeline operators to match the demand for products with the physical barrels at each loading

Symposium on Applied Computing
SAC'99, San Antonio, Texas
Copyright 1998 ACM 1-58113-086-4/99/0001

terminal. A primary goal is to minimize product outages at terminals as efficiently as possible. Pipeline companies today employ significant human resources to produce schedules that meet their business needs. A fungible pipeline is operated much like a bank; the physical currency is interchangeable. The physical dollars you deposit are almost certainly not the same you might withdraw at a later time or place. In a fungible pipeline the products are interchangeable.

The pipeline is represented as a graph where the edges are the physical pipeline segments and the vertices are storage/pump facilities (terminals). In the model shown in Figure 1, the pipe segments are considered equal in size and volume. The Hexagons represent the Terminals where products are stored and/or loaded into transport vehicles, and where operations personnel can choose to store in local tankage or pump/re-direct incoming product to downstream connecting terminals. The product storage tanks are represented by the triangles. In our test cases, the triangles are divided into two equal halves representing two different product storage tanks. In the real world these products might be differing grades of unleaded gasoline, fuel oils, diesel or aviation fuels. They are tested for various minimum quality standards upon nomination into the pipeline. Once within the pipeline system, volumes are similarly fungible. It is the goal of the pipeline schedulers to match the demand for product with the physical barrels at each location. Their challenge is to constantly adjust the location of product to match the logical barrels and the historical demand (outflow) at each terminal. Pipeline schedulers must minimize product outages at terminals, in the sense that a branch of a bank would not wish to run out of cash on hand. This problem was motivated by the real world pipeline scheduling problem from a local pipeline company.

In addition, all product movements are considered to take an equal amount of time and the minimal unit of volume is the entire linefill volume. Our model uses two product types. We allow unidirectional flow only.

Our model consists of eight terminals and eight edges, one with two connecting edges and one with three. See Figure 1. The flow moves from left to right only. The terminals have a simulated storage capacity of 0, 1 or 2 units of volume (zero is empty, one is half full, and two is a full tank). For example, in Figure 1, terminal three's storage tank has one unit of Product A (half full), and a full tank of Product B (indicated by a two). The integer nature of the value is a deliberate choice, made to limit the number of combinations. The initial storage values for linefill and product volumes in Figure 1 are arbitrary, excepting that care was taken to represent reasonable conditions.

2. APPLYING A GENETIC ALGORITHM TO PIPELINE SCHEDULING

Genetic Algorithms have been very successful in providing a method of obtaining near-optimal solutions to many combinatorial optimization problems. First described by Holland[5] in the mid 1970's, the genetic algorithm is a powerful search and optimization technique. The pipeline-scheduling problem has a very large solution space. The number of possible schedules is a function of the size of the pipeline and the number of different products it transports. A sufficiently large pipeline, transporting many different products, will produce a solution space far too large to exhaustively search. The GA approach was selected as a likely method for producing effective results[3]. Other researchers have investigated the benefits of solving combinatorial optimization problems using genetic algorithms. Davis[2], Goldberg[4], Rawlings[7], and Michalewicz[6] provide excellent examples. Our GA was implemented using LibGA[1].

To map a GA to the problem at hand, the chromosome must be developed in a way that allows the GA to function effectively. The strategy selected here was to evolve a fixed set of actions (or rules) for all possible combinations of incoming product and current storage levels for each product (rules). The terminal product storage levels are limited to a trinary state (0,1,2). The alleles or genes of the chromosome are binary values (bits) that correspond to rules and actions for each rule applied to each decision point along the pipeline schedule (that is, at each pipeline terminal). A rule is needed for all combinations of incoming product types and current storage levels for each product. A rule is needed for each possible state of the pipeline. For each of these rule combinations, an action is evolved for *each* outward-connecting pipe.

In a two product pipeline application in our model, the incoming product at each terminal could be product A, or product B, or no product. The storage volume for

product A at a terminal could be 0, 1, or 2. The storage volume for product B at a terminal could also be 0, 1, or 2. Thus there exists 27 possible combinations of the status at each terminal. The rules (or actions) once evolved for each of the 27 status combinations, are constant, and are applied to the pipeline in a series of time-steps. The application of the rules at each time-step is the resulting schedule. The fitness function of the GA is responsible for this application and evaluation.

The possible actions that can be taken at a terminal are: (a) pass the incoming product on to an outgoing connection pipe, and do nothing else, (b) pass the incoming product on to an outgoing connecting pipe, and in addition send product A to another outgoing connecting pipe, (c) pass the incoming product on to an outgoing connecting pipe, and in addition send product B to another outgoing connecting pipe, (d) store the incoming product and pass on nothing, (e) store the incoming product and pass on product A to an outgoing pipe, and (f) store the incoming product and pass on product B to an outgoing pipe. Note at some terminals some of the options are infeasible.

Since the rules and actions are held constant throughout the life of the scheduling period, this means the rules cannot change from one time-step to another time-step. This appears to be a weakness in this implementation; however, this can be corrected by adding more bits to the rule set. The additional rules would include information about the need for products at each of the possible downstream terminals. One could also solve this problem by replicating the entire chromosome as many times as the number of time-steps.

The chromosome that maps a two product instance of the pipeline identical to Figure 1 is constructed by taking the 27 possible rules for each of the 8 terminal nodes times 3 Action bits for each of the outward connections at a node.

For Figure 1 this becomes $27 \times 8 \times (3 + 3 + 9 + 6 + 0 + 0 + 0 + 0) = 4,536$ bits.

This is a large number of bits for a typical GA application. It becomes apparent that a larger pipeline with additional products will cause the chromosome to grow exponentially.

3. THE GA CONFIGURATION PARAMETERS

The following configurable parameters were used in the GA pipeline scheduling application: The chromosome was a 4536 element **bit** string. The initial

population was randomly generated, and a pool size of 100 was used. The type of GA selected was the generational model. The selection operator was the “roulette wheel” approach. The uniform binary crossover was used. The mutation strategy employed was a simple bit toggle at a rate of 0.1 percent. The replacement method appended members to a new pool. Elitism (to ensure the best members of a pool will survive until the next generation) was used.

The initial state of the pipeline including the linefill and product storage volumes are read from a configuration file. (As represented in Figure 1). The pre-set goals array for each location is read from a second file. In our case, the goals were set to a storage fill of one for all locations and both products.

4. THE OBJECTIVE FUNCTION

The objective function in this application is required to simulate the movement of product in a liquid pipeline and then assess the relative value of the End State according to the pre-set goals. The storage and linefill changes are simulated location by location, as the appropriate rules are applied. The following is an outline of the steps used in the objective function: (1) A fresh copy of the pipeline data structure is recovered. (2) Apply the nominated volume to terminal number one. This sets the ripple effect into motion. (3) For each time-step, for each terminal/location, interpret the inherent schedule by applying the actions resulting from the rules dictated by the incoming product and current product volumes. (4) Calculate the rule offset into the chromosome and obtain the action bit values for this terminal. (5) Act on the action bits by adjusting the product volumes and line fill product types. (6) Add 10 points to the running fitness value for each successful (feasible) application of the action bits at a given terminal. (7) Make any necessary and possible fix-ups to the chromosome (in time-step one only). (8) When all locations and time-steps are complete, the product volumes are compared to the goals array and 20 additional points are awarded for each goal met or exceeded.

The optimal fitness value for the proposed problem is a function of the number of time-steps and the number of goals met. The maximum fitness at time-step i in our system is defined to be $(i * \text{eight} * 10 \text{ points}) + (\text{eight} * 2 \text{ products} * 20 \text{ points})$. Thus the maximum fitness at time-step 1 is 400, and at time-step 2 is 480, and at time-step 3 is 560, etc. This award system worked well because it produced moderate differentiation between various solutions, big enough to drive the GA toward optimal values but small enough to allow the

“roulette” selection operator to find and occasionally use weaker solutions.

5. FIX-UPS

Due to the nature of the problem there can arise action combinations that make the chromosome as a whole, infeasible. For example, leaf nodes cannot do any action except store products. Infeasible solutions arise when the action at a leaf node is anything other than store the product and pass nothing on. The leaf node action can be altered or “fixed” in the first time-step by correcting their value. Another possibility that turned out to be a real problem, is that a given rule may work fine for one, two or even three time-steps, but may cause an infeasible solution at a later time-step because the down-stream storage values have changed such that an impossible situation occurs. In our system, this is not correctable because the rule has already been used and cannot be deterministically changed in mid-stream. These situations, however, could easily be corrected by adding more bits to the rules that take into account the down-stream storage volumes.

Another fix-up that was implemented in our objective function was to deterministically choose the most needy neighbor to send a product when the suggested action calls for passing a product, but actual action was set to “no product”. The action was changed in time-step one for a given location if this occurred. To accomplish this fix-up, all neighboring connections were checked for the least volume of the appropriate type. If there was a tie, the first location was used. This fix-up turned out to be critical to the success of the GA. The optimal schedule that is presented later used this fix-up.

In addition to fix ups and the detection of infeasible solutions, the main responsibility of the objective function was to assign a fitness value to each chromosome. The better the resulting schedule the ‘higher’ the fitness value. To simplify the process and to allow proactive flexible control of the schedule’s parameters, a goal table was defined before the GA was run. At each pipeline terminal and time-step, a target volume (0,1,2) for each product was entered into the “goals” data table. The chromosome fitness value was penalized for missing the target on the low side and a higher penalty assessed for letting a terminal volume go to zero. On the positive side, a bonus was given if the targets were met exactly. In this model new product is “nominated” into the pipeline at location #1 only. Product is removed from the terminals at a constant demand rate, determined before runtime. This simulates the loading of tanker trucks, rail cars and other types of product outflow from the

pipeline. In this GA implementation, a simple function was used to determine if it is the correct time-step to decrement the integer storage value for each product.

The evaluation function required that the pipeline schedule be played out in a step-by-step fashion, moving linefill to the next location as directed by the rules and adjusting the current storage values as required. This procedure was also used to produce an intelligible schedule from a 4536 bit chromosome. The GAs best solution must be translated to a format understandable to the humans controlling the pipeline.

As you recall, the problem definition includes a goals table that specifies the minimum product volume targets at each location. The goal in this case was to avoid occurrences of zero volume. The GA converged in thirty-one generations. The GA completed in 59.06 seconds. This is encouragingly fast. However, more complex problems will grow exponentially larger, and the need for CPU resources will grow as well. This solution as it turns out was optimal for our small example. This optimal solution included five product movements and the elimination of two zero volume conditions. A detailed location-by-location trace of the optimal solution is summarized below. These instructions would be sent to the terminal operators to coordinate the tank and valve alignments. Figure 1 shows the initial state of our system, and Figure 2 shows the results of applying our GA. That is, if one starts with Figure 1 and follows the instructions below then Figure 2 is the result.

A condensed version of the pipeline schedule produced by the GA:

- Location # 1 - Store incoming Product A in tankage
- Location # 2 - Send Product A from tankage to Location # 3
- Location # 3 - Pass incoming Product A to Location # 6. Send Product B from tankage to Location # 4
- Location # 4 - Pass incoming Product B to location # 8. Send Product A from tankage to Location # 7
- Location # 5 - Do Nothing
- Location # 6 - Store incoming Product B in Tankage
- Location # 7 - Store incoming Product A in Tankage
- Location # 8 - Store incoming Product B in Tankage

6. CONCLUSIONS AND FUTURE ENHANCEMENT STRATEGIES

The goal of this research is to partially or fully automate a complex process that today takes hundreds of hours of labor per week to perform in the real world. Of course the real-world pipeline-scheduling problem is more complex. Examples of additional real-world complexities include: (a) Reversible flow on some pipes. (b) Seven or more distinct product grades. (c) Variable batch sizes, tanks sizes and non-uniform time increments. (d) Multiple lines between some terminals. (e) Interface issues, (where product batches come into contact in the pipe). (f) Real-time changing of product scheduling goals as shippers move logical volumes around the pipeline. (g) Occasional emergency situations such as a pump failure. (h) Weather related demand (severe cold for extended periods). (i) The existence of multiple origin points (connections to other pipelines and refineries).

The basic concept of a rule based GA presented in this paper can (in theory) be extended to cover most if not all of these complexities listed above with little additional effort. The most notable effect will be the exponential growth in the size of the chromosome. The chromosome length is proportional to the number of rules, and the number of rules is an exponential function of the number of products and terminals. The size of the chromosome for a moderately large real-world application of 25 terminals, 7 distinct products at 16 levels of storage volume resolution, would be calculated as follows:

$25 \text{ terminals} * (\text{Rules}) * (\text{Action Bits})$, where $\text{Rules} = (3 \text{ Product bits}) * \text{Volume States (for example)} * 7 \text{ products}$. The $\text{Action Bits} = (4 * \text{Average \# of pipeline connections/ terminal})$. This produces a total of $25 * (2 \text{ to the } (3 + 28 + 6) \text{ bits})$ which is 8.5899×10 to the 11th power. Thus over 85 billion bits are needed in the chromosome. This translates to 1.0625 GB of storage.

An enhanced objective function will be required to better evaluate the subtle differences between end-states, (i.e. how well the goals are met, and position of linefill products). Is a completely full tank better or worse than one with room to place product into? Therefore the heuristics of an enhanced objective function will likely need to take into account additional factors other than minimally meeting a volume target. Alternately, taking a completely different approach, a chromosome might be developed that contains the product movement information without the need for all possible combination of incoming product and pipeline volume states. Alternatively, if all states are only

potentially needed, it is possible that a sparse matrix technique or an adaptive process that dynamically creates the needed portions of the chromosome could be used to reduce the physical chromosome storage requirement. Given the impossible storage requirements for industrial size problems, we are currently looking into some of these alternate approaches for solving this problem.

This problem was motivated by the real world pipeline scheduling problem from a local pipeline company where the first author is employed. We have no previous work from the literature to compare our algorithm or results to. However, our application of pipeline scheduling using Genetic Algorithms was successful. It produced the optimal schedule for our small example problem. The significance of this research is (1) this problem, to the authors' knowledge, has not been solved before using evolutionary computation methods or any other method other than brute force or by applying previous experience, and (2) our GA model for constructing the chromosome and applying rules as we step through the pipeline can (in theory) be scaled up for a more complicated (industrial size) pipeline network which includes more products, terminals, edges and links. However, this will require some additional work and perhaps alternative strategies due to the large storage requirements. Although this application was a simplification of the real-world pipeline scheduling problem, it does prove the concept of using a GA to solve this class of problem. By extending the number of products, number of terminals and increasing the resolution of the tank storage volumes, with care our solution can move toward an industry solution that can be applied in the commercial world.

Acknowledgements

Special thanks to Professor Tom Haynes for assisting us with the details of setting up our rule based scheduling system.

References

- [1] A.L. Corcoran and R. L. Wainwright, "LibGA: A User-friendly Workbench for order based Genetic Algorithm Research", *Proceedings of the 1993 ACM/SIGAPP Symposium on applied Computing*, pp. 111-118, 1993, ACM Press.
- [2] L. Davis ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.
- [3] K.A. DeJong and W.M. Spears, "Using Genetic Algorithms to solve NP-Complete Problems", *Proceeding of the third international conference on Genetic Algorithms. June, 1989* pp. 124-132.
- [4] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989
- [5] J. H. Holland "Adaptation in Natural and Artificial Systems", Ann Arbor: The University of Michigan Press, 1975
- [6] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 3rd edition, 1996.
- [7] G. Rawlings, editor, *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991.

Figure 1
Initial Pipeline

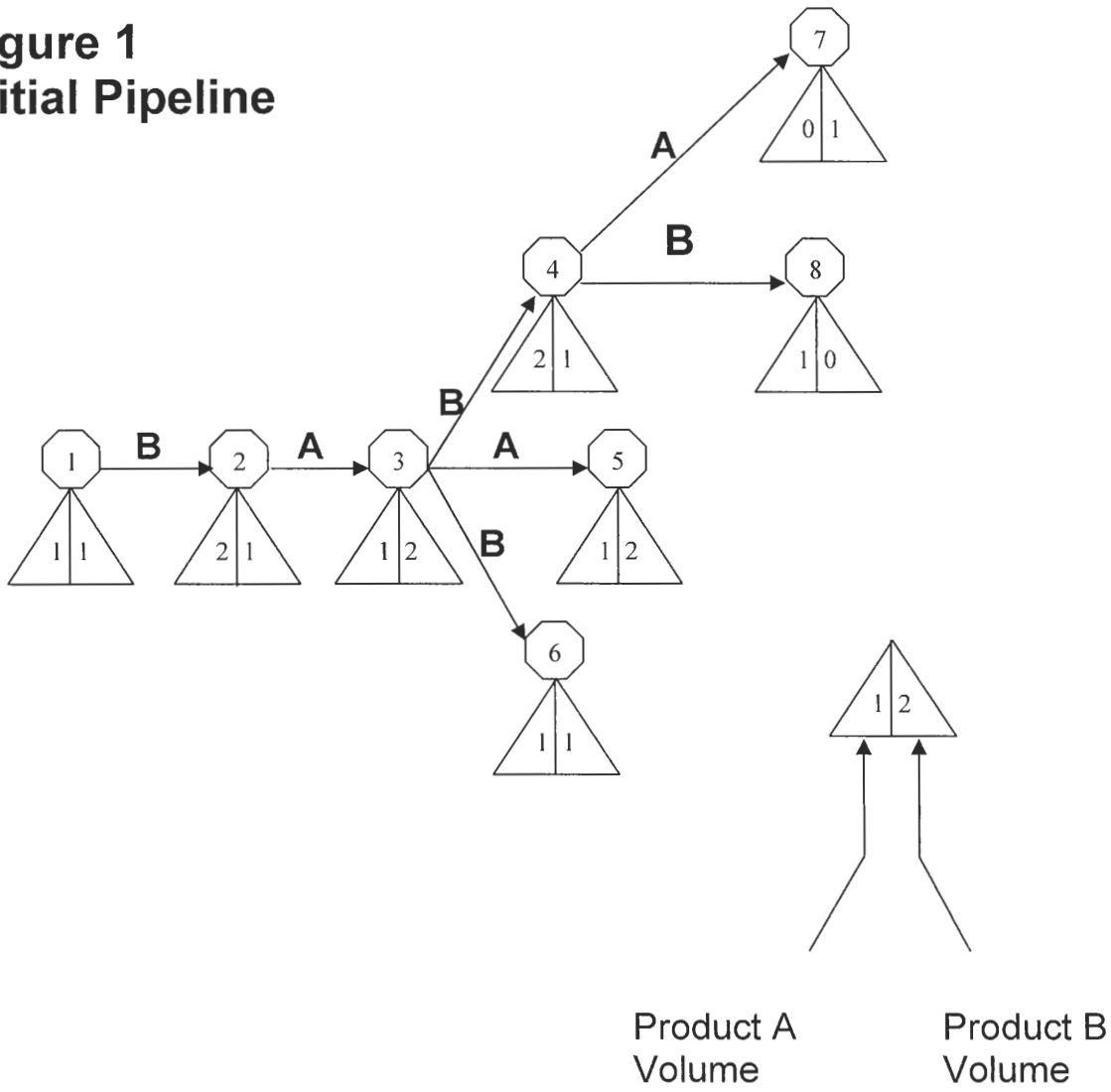


Figure 2
Pipeline After GA Schedule
Is Applied

