Drum Scheduling:

A Solvable Case of The Traveling Salesman Problem

Michael Gabilondo

COP 5537

December 7, 2009

Abstract

The Drum Scheduling problem is a type of disk scheduling problem that can be modeled as a special case of the traveling salesman problem. We discuss the disk scheduling problem and its graph model representation, as well as some possible solutions for this graph problem. In particular, we present an optimal disk scheduling $O(nlogn)$ algorithm [8], where n is the number of records to be accessed. A brief history of the traveling salesman problem and some of its more general solutions or approximations are also shown. The java software implementation of the $O(nlogn)$ solution is presented.

1 Introduction.

Drum memories consist of an elongated cylinder with an attached column of read/write heads all along the height of the cylinder (See Figure 1, taken from [8]). On the surface of the drum, and running perpendicular to the read/write heads, are the tracks of the drum. Tracks contain records, and each track is accessed by only one of the heads. The drum rotates in one direction at a constant rate, while the read/write heads remain stationary. It is only when the desired record to read or write comes under one of the heads that the record can be accessed. After accessing one record, and moving on to another record on the disk, the drum has to rotate until the desired next record comes under the heads. The sum of these time intervals that amass from traveling from one record to the next is called the *total latency time*.

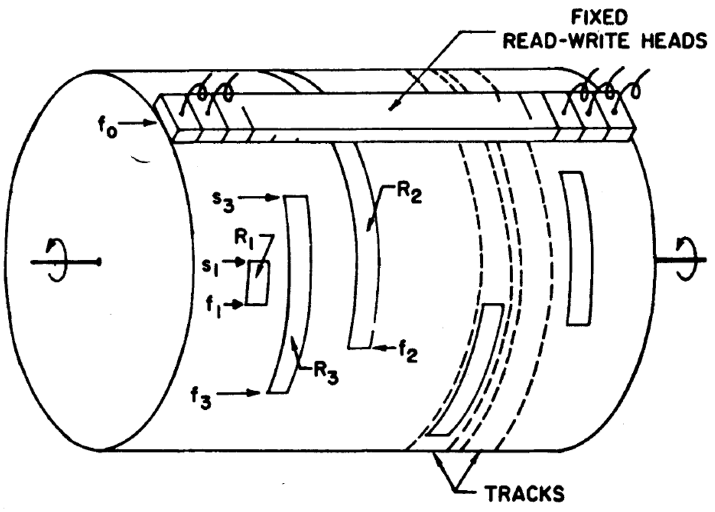


Fig. 1. Drum storage unit.

The Drum Scheduling problem is a type of disk scheduling problem. Disk scheduling is the problem whose input is a set of n indexes of records on a drum memory. The output of the algorithm is an order in which to visit the $n$records; that is, the output is a permutation of the sequence $1,2,...,n$. This is an NP-Hard problem involving searching the space of all possible permutations for the optimal permutation. Modeled as a graph problem, it is the problem of finding a Hamiltonian path in a graph of n vertices; that is, finding a path through all n vertices that only visits each vertex once and only once. This problem is also known as the traveling salesman problem [2].

As an example of a drum scheduling approach, consider Shortest Latency Time First (SLTF). This is a greedy approach in which the next record to be accessed after reading some record is the record which the read-write heads can get to fastest. In other words, it chooses the record that will cause the traveling time between the current record and next record to be minimized [5]. SLTF is not optimal since the records overlap. If the records did not overlap, choosing the next record to access as the record with the shortest latency time would be optimal. But since the records overlap, the decision of reading one record means that during the process of reading the record, the beginning of an overlapping record moves past the read-write heads without getting accessed.

We present an algorithm [8] that exploits the special structure of the disk scheduling problem to find an optimal solution in $O(nlogn)$ time. The basic approach is to find a set of disjoint cycles of the records. Each cycle represents some schedule, but the solution needs to have only one schedule that involves all of the records. In [8], the sum of the latencies of all of the schedules is proven to be the the minimum out of all possible sets of schedules. If the number of cycles (schedules) is greater than 1, then the cycles are "patched" together by interchanging edges. It is also proven in [8] that the algorithm always performs the sequence of edge interchanges with minimum cost out of all possible sequences of edge interchanges. Hence, it is shown that the schedule found by the algorithm is optimal.

The next section gives some background behind the traveling salesman problem. Section 3 discusses the problem of drum scheduling and potential solutions and presents an optimal drum scheduling algorithm [8] that runs in $O(nlogn)$ time. Section 4 discusses a software implementation of the optimal algorithm. Section 5 concludes with a short discussion.

2 Traveling Salesman Problem.

An example instance of the traveling salesman problem can be stated as follows. A traveling salesman living in Orlando, FL would like to go on a tour of the United States to peddle his wares. He wants to visit 100 of the major cities, but wants to minimize the amount of money or time spent traveling, and wants to return to Orlando at the end of the tour. Visiting a city more than once is unacceptable. He wants to know the best order in which to visit these cities. A solution to this problem is a permutation of these 100 cities, such that the desired quantity (such as money or time) is minimized.

The traveling salesman problem can be modeled as the problem of finding a Hamiltonian cycle in a graph with the cheapest cost. We are given a graph G with n vertices, and m weighted edges, where the edge from vertex i to vertex j has cost $c_{i,j}$. The symmetric traveling salesman problem involves only undirected edges, where $c_{i,j} = c_{j,i}$, for any vertices i and j. The asymmetric problem involves directed edges, so that in general $c_{i,j} \neq c_{j,i}$. A solution to the problem is a Hamiltonian cycle with cheapest cost, which means that a solution must meet both of the following criteria [2]:

- **Hamiltonian cycle**. The path must be through all n vertices, such that each vertex is visited only once, except for the first and last vertices, which must be the same.
- **Cheapest cost**. The Hamiltonian cycle chosen must be one in which the sum of the costs of the edges is minimized. There may be more than one Hamiltonian cycle with smallest cost.

An obvious brute-force approach is try all permutations of n vertices and choose the one which satisfies both of the properties above. This involves trying $\Theta(n!)$ possibilities and is not computationally feasible for all but very small graphs. Early attempts at exact solutions involved linear programming and branch-and-bound techniques [9], but the solutions were still exponential. More recently, approaches involving cutting planes have solved instances with over 85,000 cities [1].

There are also approximation algorithms in which heuristics are used to obtain a near-optimal solution in less than exponential time. For example, some approximation heuristics for the traveling salesman problem *in the plane* involve subdividing the cities into small groups, finding optimal solutions for these small groups and then reconstructing the solution for the entire planar graph [4] [6]. As an illustration, consider Figure 2. On the left, the planar graph is divided into regions and optimal or heuristic solutions are found for the regions. On the right, the regions are merged together into one graph to obtain an approximate solution.

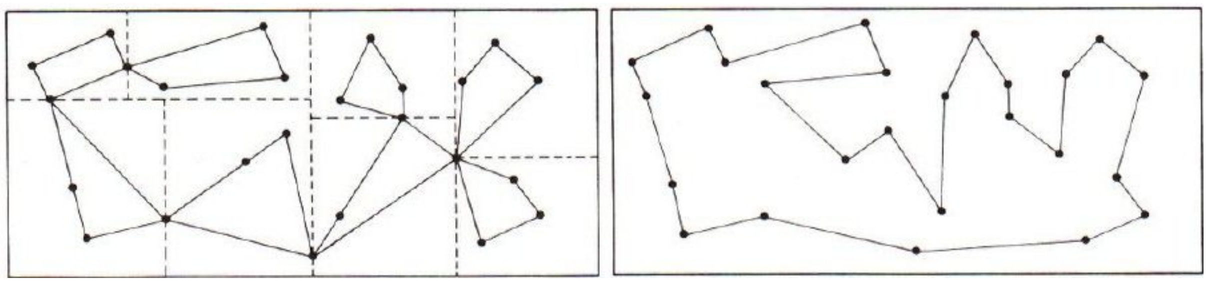


Fig 2. Constructing an approximate solution to traveling salesman problem on a planar graph. Taken from [6].

Besides the general form of the traveling salesman problem, there are also special cases that are exactly solvable in polynomial time. Sections 3 gives an overview of one such special case: the drum scheduling problem. Sections 3 and 4 describe the $O(nlogn)$ algorithm and software implementation.

3 Drum Scheduling Problem.

The drum scheduling problem, presented first in the Introduction, is the problem of finding the "best" order in which to process n records on a drum. In this paper, "best" means the minimal total processing time (MTPT) discipline. In the MTPT, the goal is to minimize the total latency resulting from processing n records [7]. The SLTF discipline results in near-optimal MTPT, which is never more than one full drum revolution more than the minimal time [5]. Another related approach is to consider drum scheduling as an instance of the Minimum Latency Problem, also known as the traveling repairman problem [3]. In this problem, the average total time each record must wait to be processed is minimized, rather than minimizing the sum of all of the latency times.

Figure 3 (taken from [8]) shows the surface of the drum with three tracks shown. The first track has record R_1, the second track as record R_2 and the third track has record R_3. R_1 starts at s_1 (0.2) and ends at f_1 (0.3). In general, record R_i starts at s_i and ends at f_i. The numbers on the left running vertically are the positions on the drum. For example, if the read-write heads are at position 0.5, then they would have access to the beginning of R_2 (s_2) and to the center of R_3. With respect to Figure 3, the heads travel upwards at a constant rate, and when they reach position 1.0, they are immediately reset to position 0.0, since these two positions are actually the same position on the drum. It takes 1 unit of time and 1 unit of length for a single rotation. Note that even though the heads remain stationary, it is possible to view them as moving across the drum with the above model.

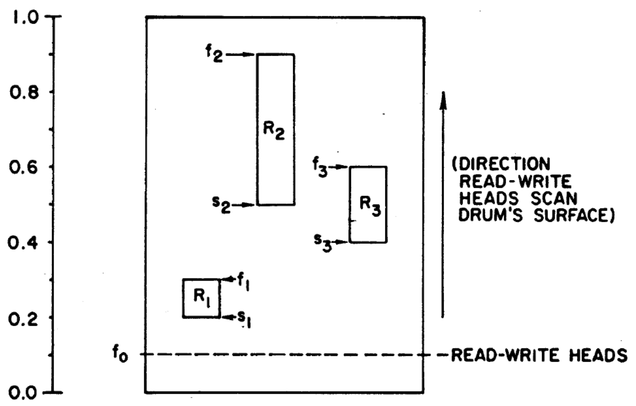


Fig. 3. Location on the drum's surface of the three records and the initial position of the READ–WRITE heads.

The time it takes for the heads to move from f_i to s_j, where record i was just accessed and record j will be accessed next, is called the latency time from record i to record j. If $f_i \leq s_j$, then the latency time is defined as $s_j - f_i$. If $f_i > s_j$, then the latency time is $1+(s_j - f_i)$. The initial position of of the read-write heads is denoted f_0, which corresponds to the end of a special made-up record called a *pseudorecord* (also called R_0). The beginning of the pseudorecord is denoted s_0. For convenience, the problem will be modeled as finding the cheapest schedule of $n+1$ records, where n is the number of real records and the other one is the pseudorecord.

Any valid schedule must start with R_0, since the end of R_0 is defined to be the beginning position of the read-write heads. Hence, the latency between R_0 and any other record R_k is $s_k - f_0$ if $s_k \geq f_0$ or $1+(s_k - f_0)$ if $s_k < f_0$. By defining the end of the pseudorecord as the start of any record, the latency between any record and R_0 equals 0. This zero cost also allows us to model the problem as finding a cycle through the $n+1$ records rather than a path through the real n records, since returning to the pseudorecord from the last record does not change the total latency time.

Define ϕ as a set of disjoint permutations, where each permutation uses a subset of the numbers from $\{0,1,...,n\}$, as in $\phi = \{(0,3,5),(1,2,4)\}$ for $n=5$. The solution to this problem is the space of all ϕ that have exactly one permutation, and the restriction that the single permutation it contains starts with 0. The total latency or cost of ϕ is

$$C(\phi)=\sum_{i=0}^{n} t_{i,\phi(i)}$$

where $\phi(i)$ represents the successor of record i in the set of permutations ϕ, and $t_{i,\phi(i)}$ is the latency time from record i to record $\phi(i)$. For example, the non-optimal SLTF schedule of Figure 1 is $(0,1,3,2)$. The cost of this schedule is $t_{0,1}+t_{1,3}+t_{2,0}+t_{3,2}=1.1$. While any ϕ with more than one permutation is not a legal solution, this broader definition is used in the intermediate steps to the solution.

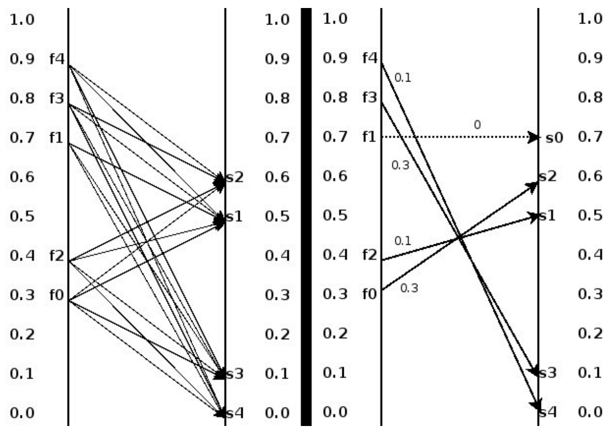


Figure 4. (Left) problem instance, (Right) result of minimum matching procedure.

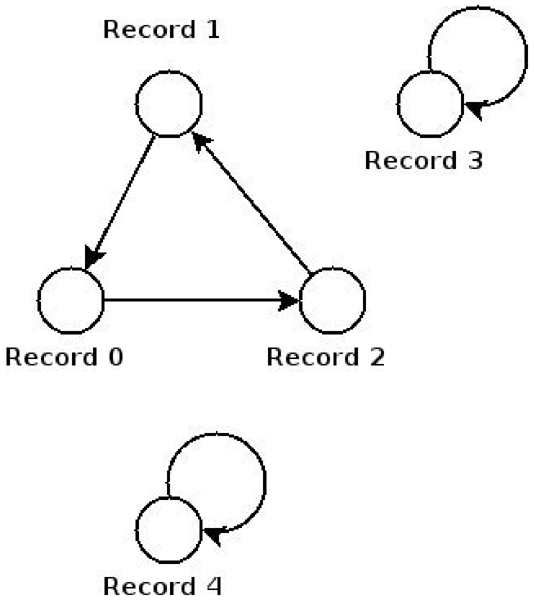


Figure 5. Three disjoint schedules corresponding to Figure 4 (Right).

The problem can be represented graphically using an *ordered bipartite graph,* as in Figure 4 (Left). It is ordered because the labels are positioned somewhere along [0, 1]. The only edges are from f_i to s_j. Figure 4 (Left) represents a complete ordered bipartite graph, where every possible edge is present. An edge from f_i to s_j means to process record j after record i. The goal is to choose a subset of the edges that represents a single cycle minimum cost permutation. If an edge from f_i to s_j is chosen, then any other edge incident on f_i or s_j cannot be chosen. In the rest of the paper, sometimes a symbol f_k will be meant as that node itself, and other times it will be meant as the position of that node in the ordered bipartite graph (for example, 0.3), and which one should be clear by the context.

The general approach behind the algorithm is to first generate a ϕ that is a set of possibly disjoint permutations, by minimizing $C(\phi)$, ignoring the restriction that ϕ must contain a single cycle. For example, consider the set of two disjoint permutations, $\{(0,3,5),(1,2,4)\}$. The first permutation represents a cycle or schedule of records 0, 3 and 5, and the second of records 1, 2 and 4. Following this step, a sequence of *edge interchanges* is performed; the result of each interchange is to join two disjoint cycles. In the example, the first schedule has the edges $f_0 \to s_3, f_3 \to s_5, f_5 \to s_0$, and the second schedule has edges $f_1 \to s_2, f_2 \to s_4, f_4 \to s_1$. An edge interchange between two edges swaps the successor nodes of the two edges. For example, interchanging the first edge of the first permutation with the first edge of the second permutation removes the two existing edges and creates two new edges, $f_0 \to s_2, f_1 \to s_3$, which creates the permutation $(0,2,4,1,3,5)$. Note that an edge interchange can change the cost of the disjoint set of permutations ϕ, but a simple look-up table is defined in [8] and it is easy to tell whether an interchange will result in negative, positive or zero cost.

The algorithm is now described in more detail. The first step is to minimize $C(\phi)$, ignoring the restriction that ϕ must contain a single cycle. This is done by first performing the greedy *minimum matching procedure* on the complete ordered bipartite graph. This procedure simply chooses the set of edges with the smallest costs, such that no two edges chosen are incident on the same node. It is proven in [8] that this ϕ minimizes $C(\phi)$. Figure 4 (Right) shows the result of the minimum matching procedure executed on Figure 4 (Left), and Figure 5 shows the schedules or cycles corresponding to the matching. Note that Figure 4 (Left) has 5 f_i labels but only 4 s_j labels, since there is no s_0 label on the right. Therefore, at the end of the minimum matching procedure, there will always be some f_δ not incident on any edge. The latency time between any record and s_0 is defined to be 0, so the edge $f_\delta \to s_0$ can be added at the end of the matching.

Next, several transformations that do not alter the total cost are applied to the resulting graph. These transformations are used make it easier for [8] to prove the optimality and correctness of the algorithm. The first such transformation is to shift the point of reference on the drum to f_δ. This changes the positions of the nodes on the ordered bipartite graph, but does not change the relative positions of the nodes. f_δ is moved to position 0, and the position of node z is changed to $z - f_\delta$ if $z \geq f_\delta$ and $1 + (z - f_\delta)$ otherwise. At this point, any "crossed" edges can also be uncrossed in any order, without affecting the cost of ϕ, since it is shown that any such edge uncrossing at this point is a type of zero cost edge interchange. Following the running example, at this point the graph looks as shown in Figure 6. An interchange of edges $f_i \to s_j$ and $f_p \to s_q$ has zero cost if and only if $f_i, f_p \in [s_j, s_q)$ or $f_i, f_p \notin [s_j, s_q)$. For example, in Figure 6, (f_2, f_0) and (f_4, f_3) define zero cost interchanges but (f_0, f_4) and (f_3, f_1) define positive cost interchanges, since they increase the total cost of the edges.

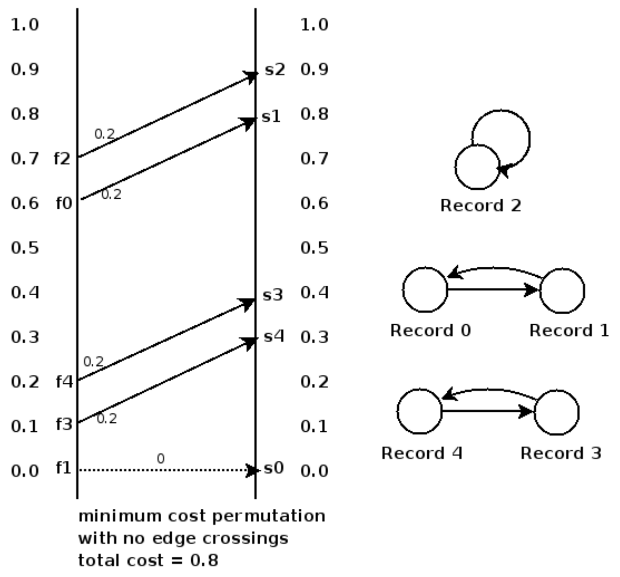


Figure 6. On the left is the output of the minimum matching procedure after the point of reference was shifted $f_1 = f_\delta$ and then edges were uncrossed. On the right are the corresponding cycles.

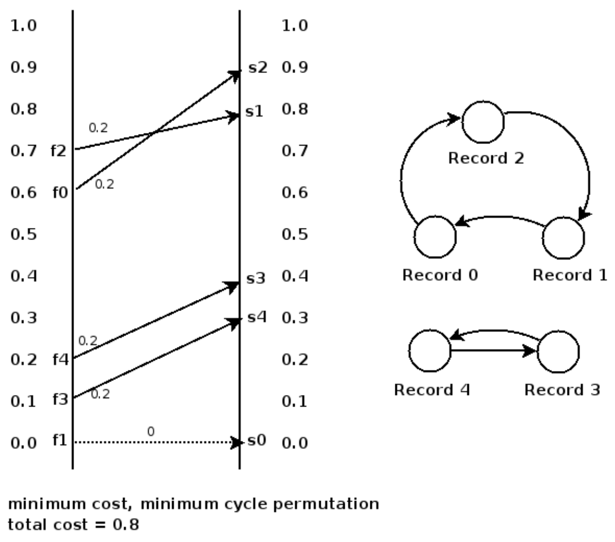


Figure 7. On the left is the output of the series of zero cost interchanges applied to Figure 6. It has the minimum number of cycles and minimizes $C(\phi)$.

The next step performs a series of zero cost interchanges to minimize the number of disjoint cycles in ϕ. It is to check each pair of *adjacent* nodes (f_i, f_j) from top to bottom, and if they are in different cycles and their respective edges define a zero cost interchange, then do the interchange. Two nodes f_i and f_j are adjacent if there is no other node f_k such that $f_i \le f_k \le f_j$ (it is assumed $f_i < f_j$). The result of this step is shown in Figure 7. It turns out that after this step, we have the set ϕ with the minimum number of cycles that minimizes $C(\phi)$, ignoring the restriction that ϕ must contain a single cycle.

If there is only one cycle after this, then the algorithm is done. However, it is proven in [8] that at this point there are no zero or negative cost interchanges to reduce the number of cycles if the number of disjoint cycles is greater than one. Furthermore, they show that a specific sequence of positive cost interchanges will transform ϕ to minimize $C(\phi)$, *with* the restriction that ϕ contains only one permutation. Recall that f_δ is defined as that node which has an outgoing edge to s_0. By repeatedly interchanging the edge incident with f_δ with the edge incident with some f_k, such that $f_k > f_\delta$ and records k and δ are in different cycles, the total cost will increase by f_q . The final graph and schedule is shown in Figure 8.

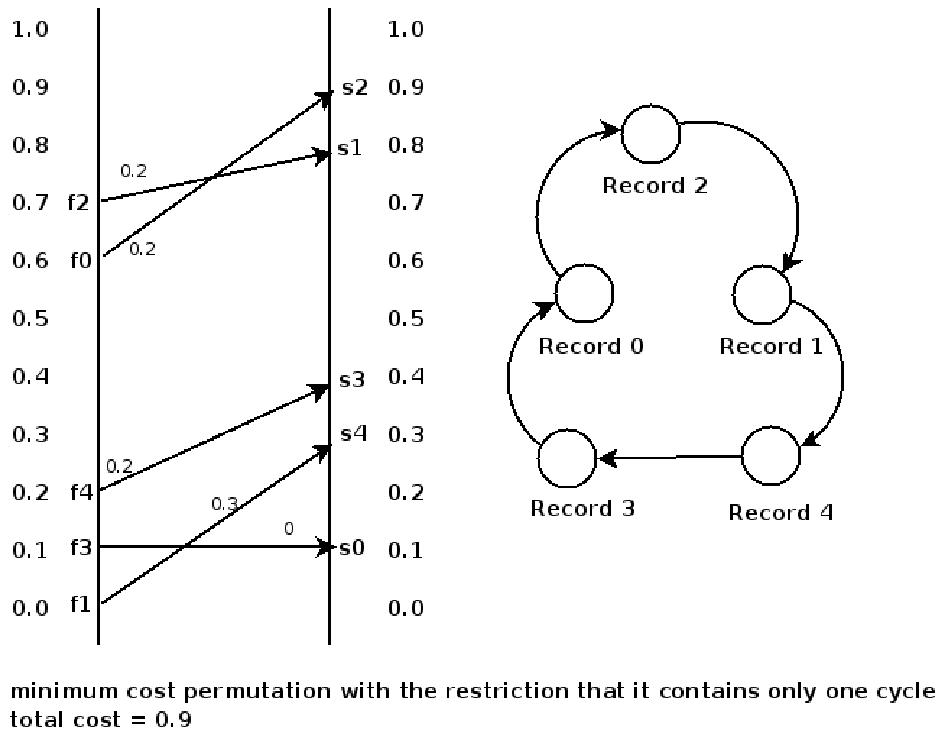
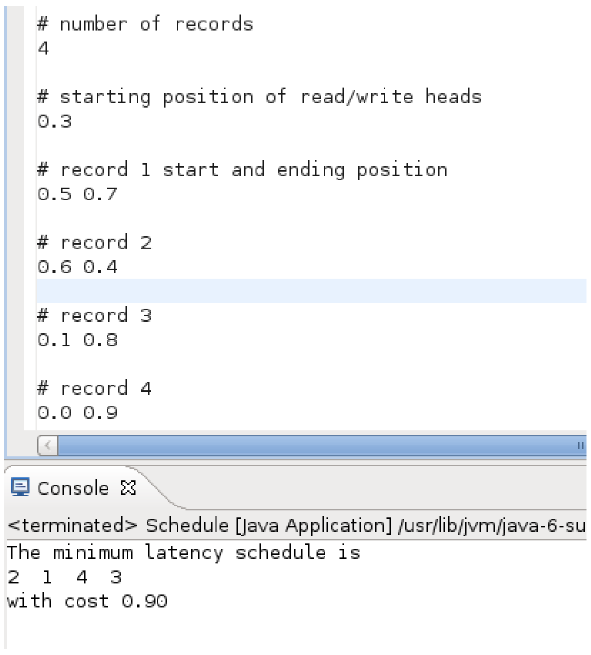


Figure 8.

4 Algorithm Implementation.

The optimal scheduling algorithm described in Section 3 was implemented in Java 6.0. The minimum matching procedure was implemented by sorting the nodes and then using a simple greedy $O(n)$ procedure using a stack. A permutation of $n+1$ records was represented using an array A of $n+1$ integers, where A_i corresponds to $\phi(i)$. After the minimum matching procedure, it was necessary to track whether any two records belonged to the same cycle. For this, I implemented a disjoint-sets data structure with the union and find operations. A screen capture of the program running in an IDE can be seen in Figure 9.



```
# number of records
4

# starting position of read/write heads
0.3

# record 1 start and ending position
0.5 0.7

# record 2
0.6 0.4

# record 3
0.1 0.8

# record 4
0.0 0.9
```

```
<terminated> Schedule [Java Application] /usr/lib/jvm/java-6-su
The minimum latency schedule is
2  1  4  3
with cost 0.90
```

Figure 9. A screen capture of the scheduler output and an example input file.

The algorithm can be seen to be $O(nlogn)$ since the most costly procedure is the sorting during the minimum matching procedure. After that, a constant number of passes are made over the nodes, which amount to only $O(n)$.

5 Conclusion.

We have presented the problem of drum scheduling as a solvable instance of the traveling salesman problem. An optimal drum scheduling algorithm from [8] was explained at length and implemented. We also gave a brief look at the traveling salesman and drum scheduling problems in a broader context.

It was interesting to see that such a notorious NP-Hard problem, the traveling salesman problem, could be tackled with an $O(nlogn)$ algorithm, and that we could find an exact solution. I also found the idea behind the algorithm intuitively appealing because it is a pattern seen through other traveling salesman problems: find small optimal solutions and merge the optimal solutions to obtain a new solution. Put another way, we arrive at one solution simply, and then gradually improve that solution to arrive at the final solution to the problem.

I also found that I implemented the algorithm in one or two nights, but spent some weeks understanding the paper referenced in [8]. Understanding why the algorithm is correct was harder than understanding the actual algorithm. I felt that my time dedicated in this project was imbalanced, since I concentrated mainly on a single paper, and did not get enough time as I would have liked understanding the other papers. I have also gained an understanding of how drum memories work, which I didn't even know existed before this.

6 References.

[1] Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook (2006), *The Traveling Salesman Problem.*

[2] K. Hoffman. Traveling salesman problem. http://iris.gmu.edu/~khoffman/papers/trav_salesman.html. 2000.

[3] Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., and Sudan, M. 1994. The minimum latency problem. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on theory of Computing* (Montreal, Quebec, Canada, May 23 - 25, 1994). STOC '94. ACM, New York, NY, 163-171.

[4] K. L. Clarkson. Approximation algorithms for the planar traveling salesman tours and minimum-length triangulations. In proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 17-23.

[5] H.S. Stone and H. Fuller. On the near-optimality of the shortest latency-time-first drum scheduling discipline. Communications of the ACM, 16(6) (1973), pp. 352-353.

[6] Karp, Richard M. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane. MATHEMATICS OF OPERATIONS RESEARCH 1977 2: 209-224

[7] Fuller, S. H. 1974. Minimal-total-processing time drum and disk scheduling disciplines. *Commun. ACM* 17, 7 (Jul. 1974), 376-381.

[8] Fuller, S. H. 1972. An Optimal Drum Scheduling Algorithm. *IEEE Trans. Comput.* 21, 11 (Nov. 1972), 1153-1165.

[9] Held H, Karp RM. The traveling salesman problem and minimum spanning trees. Operations Research 1970; 18:1138-62.

[10] Gilmore, P. C., Gomory, R. E. Sequencing a One State-Variable Machine: A Solvable Case of the Traveling Salesman Problem OPERATIONS RESEARCH 1964 12: 655-679