UNLV | University Libraries
University of Nevada, Las Vegas

UNLV Retrospective Theses & Dissertations

1-1-2008

A study of Monge matrices with applications to scheduling

Revanth Pamballa
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Pamballa, Revanth, "A study of Monge matrices with applications to scheduling" (2008). *UNLV Retrospective Theses & Dissertations*. 2421.
https://digitalscholarship.unlv.edu/rtds/2421

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

A STUDY OF MONGE MATRICES WITH APPLICATIONS TO SCHEDULING

by

Revanth Pamballa

Bachelor of Technology in Computer Science and Information Technology
Jawaharlal Nehru Technological University, Hyderabad, India
August 2005

A thesis submitted in partial fulfillment of the
requirements for the

Master of Science Degree in Computer Science
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2008

UMI Number: 1463524

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1463524

Copyright 2009 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

December 01, 20 08

The Thesis prepared by

Revanth Pamballa

Entitled

A Study of Monge Matrices with Applications to Scheduling

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science.

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

A Study of Monge Matrices with Applications to Scheduling

by

Revanth Pamballa

Dr. Wolfgang Bein, Thesis Advisor, Examination Committee Chair
Associate Professor, Department of Computer Science
University of Nevada, Las Vegas

In this study of Monge properties I summarize a few aspects of the rich material based on these properties. Monge properties are related to the theory of convexity and just as convexity is important in classical Mathematics so is the theory of Monge properties important in Computer Science. In this thesis I will summarize a few aspects of Monge properties. I compare some of the different properties and show how that they all go back to Gaspard Monge's original ideas. I will highlight a number of new results in scheduling and show how Monge properties and Monge-like properties play a role. The thesis will also give results that come from implementing a linear time algorithm for scheduling which is based on Monge properties. I focus especially on batching problems, which are important for TCP/IP acknowledgement.

The contribution of my thesis is a summary of important results regarding Monge as well a new implementation of a heuristic for an NP-hard scheduling problem. The main part of my program is a linear time subroutine which is based on a Monge-like property, called the "product property".

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I am greatly thankful to my thesis advisor Dr. Wolfgang Bein, for his guidance, support, patience and encouragement throughout the research work. I will always be indebted to him for introducing me to Monge matrices and in giving me a greater insight into the concept of Theoretical Computer Science, his knowledge and passion has inspired me greatly and it was a great privilege working with him.

I am fortunate to have benefited from other committee members, Dr. Ajoy K. Datta, Dr. Lawrence Larmore and Dr. Muthukumar Venkatesan. I express my gratitude for their support and guidance through the two years of my M.S. study.

CHAPTER 1

INTRODUCTION

Monge arrays get their name from the French mathematician Gaspard Monge (1746-1818) who first observed the property in the study of certain geometric problems. His noted memoir "Sur la théorie des déblais et des remblais" (Mém. de l'acad. de Paris, 1781), gave seminal results in the theory of the curvature of a surface. In more modern times in 1961 Hoffman [11] used the property to solve certain transportation problems. Hoffman showed that if the cost matrix satisfies the Monge property then the transportation problem can be solved with a greedy approach in linear time. Hoffman attached Monge's name to such arrays as the underlying idea of solving the problem comes from the idea noticed by Monge in 1781.

After Hoffman's work, Monge properties were not studied much until the 1980s when there was renewed interest and Monge properties wee studied extensively in Computer Science. Monge properties have been observed and used in computational geometry, to speed up linear programs, and in many optimization problems. Also, there are now many generalizations of Monge properties and many Monge-like properties, which are inspired by the original Monge property definition but which are not identical to the original property. In 1996, Burkhard [10] et al. published a first survey paper on Monge properties specifically for

optimization problems. Regarding dynamic programming speedup there is a Ph.D. thesis by Park [13]

In this study of Monge properties I summarize a few aspects of the rich material based on these properties. Monge properties are related to the theory of convexity and just as convexity is important in classical Mathematics so is the theory of Monge properties important in Computer Science. In this thesis I will summarize a few aspects of Monge properties. I compare some of the different properties and show how that they all go back to Gaspard Monge's original ideas. I will highlight a number of new results in scheduling and show how Monge properties and Monge-like properties play a role. The thesis will also give results that come from implementing a linear time algorithm for scheduling which is based on Monge properties. I focus especially on batching problems, which are important for TCP/IP acknowledgement.

The contribution of my thesis is a summary of important results regarding Monge as well a new implementation of a heuristic for an NP-hard scheduling problem. The main part of my program is a linear time subroutine which is based on a Monge-like property, called the "product property".

CHAPTER 2

LITERATURE REVIEW

2.1 Monge Properties:

Starting with the definition for a Monge matrix:

Monge matrix: (*Definition*): Any m × n, where m, n $\in Z^+$, matrix T is called a Monge matrix if it satisfies the following condition,

$T[i_1, j_1] + T[i_2,j_2] \leq T[i_1,j_2] + T[i_2,j_1]$, for all $1 \leq i_1 < i_2 \leq m$, $1 \leq j_1 < j_2 \leq n$.

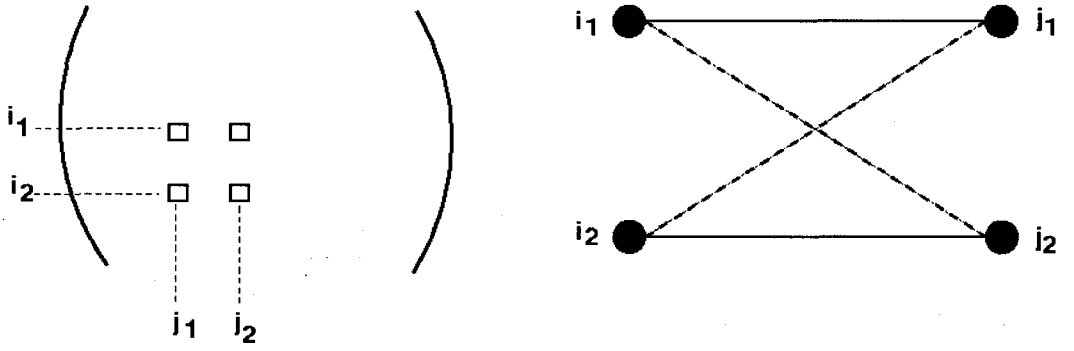


Figure 1: A 2 × 2 Sub-Matrix

In Figure 1, the right diagram shows a situation that could give rise to the entries in the matrix on the left, if the entries come from the Euclidian distance.

Equivalently, using transitivity, the following can be used as a definition for a Monge matrix:

Monge matrix (*Definition*): Any m×n, where m, n $\in Z^+$, matrix T is called a Monge matrix if it satisfies the following condition,

$T[i, j] + T[i+1,j+1] \leq T[i, j+1] + T[i+1, j]$, where $1 \leq i \leq m$, $1 \leq j \leq n$.

We note that if in the inequality above "≤" is replaced by "≥" then we call a matrix *reverse Monge*.

If we consider the rows i_1, i_2 and columns j_1, j_2 and take the elements at the intersection, the sum of the elements lying on the regular diagonal is always smaller than the sum of the elements on the reverse diagonal.

Example:

Consider the following (6 × 8) Monge array:

	j_1						j_2	
	47	46	45	44	43	42	41	40
	46	44	42	40	38	36	34	32
i_1	45	42	39	36	33	30	27	24
	44	40	36	32	28	24	20	16
i_2	43	38	33	28	23	18	13	8
	42	36	30	24	18	12	6	0

Take a 2×2 sub-matrix formed from by taking the elements at intersection of the rows i_1, i_2 and columns j_1, j_2.

42 27

38 13

The sum of the elements on the original diagonal = 55

The sum of the elements on the reverse diagonal = 65.

Clearly, the sum of elements on the original diagonal is less than the sum of the elements on the reverse diagonal.

The Monge property can be fully characterized in the following way [4]:

Given any Monge array T[i, j]: there exists vectors a[i], b[j] and a distribution array P[i, j] such that:

$$T[i, j] = a[i] + b[j] - P[i, j], \quad \text{where } i = \{1,2,3,....,m\}, j = \{1,2,3,....,n\} \text{ and } m, n \in Z^+$$

We note that a function P: N×N → R is called a *distribution function* if it satisfies

$$P[i, j] = \sum_{k=1}^{i} \sum_{l=1}^{j} p_{kl} \quad \text{for some } p_{kl} > 0.$$

The above characterization can be used to randomly generate Monge test data, by randomly generating p_{kl} values.

More generally, the Monge property can be defined for an ordered commutative semi-group (H, Δ, «) where the semi-group operation Δ is compatible with the order relation "«" i.e., for all a, b, c in the semi-group a « b implies c Δ a « c Δ b. See [5,6].

Algebraic Monge matrix (*Definition*): Any m×n, where m, n $\in Z^+$, matrix T is called an algebraic Monge matrix if it satisfies the following condition,

$$T[i_1, j_1] \Delta T[i_2,j_2] \ll T[i_1,j_2] \Delta T[i_2,j_1], \text{ for all } 1 \le i_1 < i_2 \le m, 1 \le j_1 < j_2 \le n.$$

If the operation "Δ" is replaced by the max operation we say that array T has the bottleneck Monge property.

Monge arrays can be defined in higher dimensions, see [8]:

Any array with dimension $d \geq 2$,

$n_1 \times n_2 \times n_3 \times \ldots\ldots n_d$ is said to be Monge if it satisfies the following relation

$$a[s_1, s_2, s_3, \ldots s_d] + a[t_1, t_2, t_3, \ldots t_d] \leq a[i_1, i_2, i_3, \ldots i_d] + a[j_1, j_2, j_3, \ldots j_d]$$

where $1 \leq i_k \leq n_k$, $1 \leq j_k \leq n_k$ and

$$s_k = \min \{i_k, j_k\} \text{ and } t_k = \max\{i_k, j_k\}$$

We can deduce the relation for a two dimensional Monge array from the relation for the d- dimensional Monge array. Figure 2 illustrates the above definition in three dimensions.

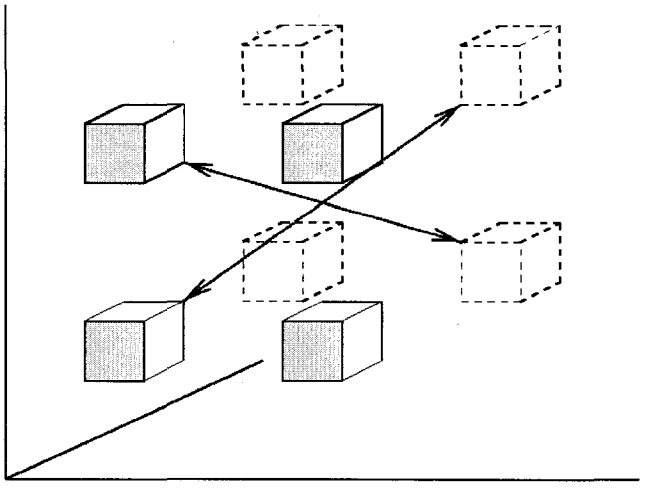


Figure 2: The 3-dimensional Monge Property

Note that in three dimensions all two dimensional planes (i.e. "all slices of the cube") are two-dimensional Monge matrices.

2.2 Examples of Common Monge Arrays:

Next we present examples of Monge arrays, with short explanations showing the Monge property:

1. $c[i, j] = i + j$

   $c[i,j]= i + j$

   $c[i_1, j_1] + c[i_2,j_2]$

   $= i_1+j_1+i_2+j_2 = i_1+j_2+i_2+j_1 = c[i_1, j_2]+c[i_2, j_1]$

Here the matrix is both Monge as well as reverse Monge; this is called "flat".

2. $c[i,j]=nm - ij$

   $c[i_1,j_1]+c[i_2,j_2] + nm - i_1j_1 + n.m - i_2j_2.$

   $L.H.S = 2nm-(i_1j_1+i_2j_2)$

   $R.H.S = 2nm -(i_1j_2+i_2j_1)$

   $i_1j_2 + i_2j_1 \leq i_1j_1 + i_2j_2$, or. $i_1(j_2-j_1) \leq i_2(j_2-j_1)$. Given $i_1 \leq i_2$ the property follows.

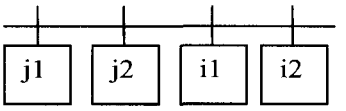
3. $c[i, j]= \max\{i, j\}$

   $c[i_1,j_1]+c[i_2,j_2] = \max\{i_1,j_1\}+\max\{i_2,j_2\} = L.H.S$

   $c[i_2,j_1]+c[i_1,j_2] = \max\{i_2,j_1\}+\max\{i_1,j_2\} = R.H.S$

Given $i_1<i_2$, $j_1<j_2$ we have to consider a number of cases:

Case 1:

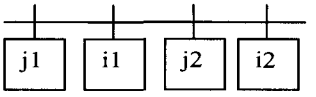


$LHS = i_1+i_2$

$RHS = i_1+i_2$

$LHS = RHS$

Case 2:

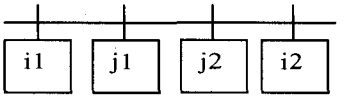


$LHS = i_1 + i_2$

$RHS = j_2 + i_2$

$LHS \leq RHS$

Case 3:

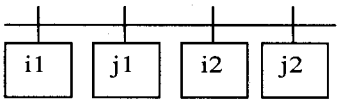


$LHS = j_1 + i_2$

$RHS = j_2 + i_2$

$LHS \leq RHS$

Case 4:

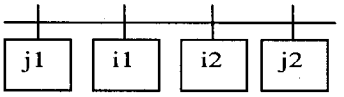


$LHS = j_1 + j_2$

$RHS = i_2 + j_2$

$LHS \leq RHS$

Case 5:

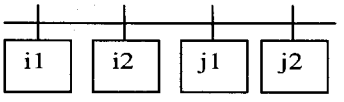


$LHS = j_1 + j_2$

$RHS = i_2 + j_2$

$LHS \leq RHS$

Case 6:



$LHS = j_1 + j_2$

$RHS = j_1 + j_2$

$LHS = RHS$

4. For a positive integer n, given are p_0, w_0, …, p_n, w_n with $p_0=0$, $w_0=0$ and p_1, $w_1 \geq 0$, …, $p_n, w_n \geq 0$, as well as s=1.

Let $P_k = \sum_{i=0}^{k} P_i$ and $W_k = \sum_{i=0}^{k} w_i$, for k=1,….n.

Consider matrix C = c[i, j], where i = 0,…,n j= 1,…,n with

$$C[i,j] = \begin{cases} (w_n - w_i)(s + p_j - p_i) & if \quad i < j \\ \infty & else \end{cases}$$

Proof of Monge property: $C[i, j] + C[i+1, j+1] \leq C[i, j+1] + C[i+1, j]$

(i, j) $(W_n - W_i)(s+P_j - P_i)$	(i, j+1) $(W_n - W_i)(s+P_{j+1} - P_i)$
(i+1, j) $(W_n - W_{i+1})(s+P_j - P_{i+1})$	(i+1, j+1) $(W_n - W_{i+1})(s+P_{j+1} - P_{i+1})$

LHS- RHS = $(-p_j)\ (w_j) \leq 0$

2.3 Monotone Matrix:

A Monge matrix is totally monotone, we define this term now:

Monotone matrix *(Definition)*: A 2×2 matrix A is called monotone matrix if $A[1,1] \geq A[1,2]$ implies $A[2,1] \geq A[2,2]$. An m×n matrix (where m, n $\in Z^{+}$) is called totally monotone if every 2×2 sub-matrix is monotone.

Obviously, a Monge matrix A is totally monotone. To see this (c.f. Figure 3) assume $A[i_1,j_1] \geq A[i_1,j_2]$ and $A[i_2,j_1] < A[i_2,j_2]$ (for some $1 \leq i_1 < i_2 \leq m$, $1 \leq j_1 < j_2 \leq n$). But this would imply $A[i_1, j_1] + A[i_2,j_2] \geq A[i_1,j_2] + A[i_2,j_1]$. However not every totally monotone matrix is Monge.

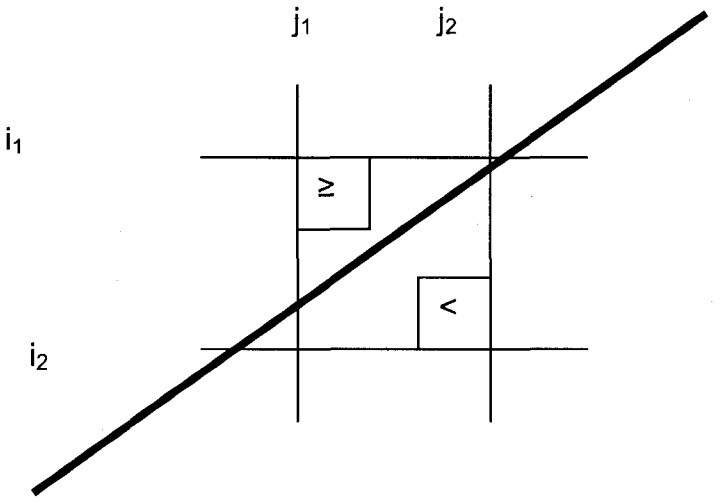


Figure 3: Monge Matrix is Totally Monotone

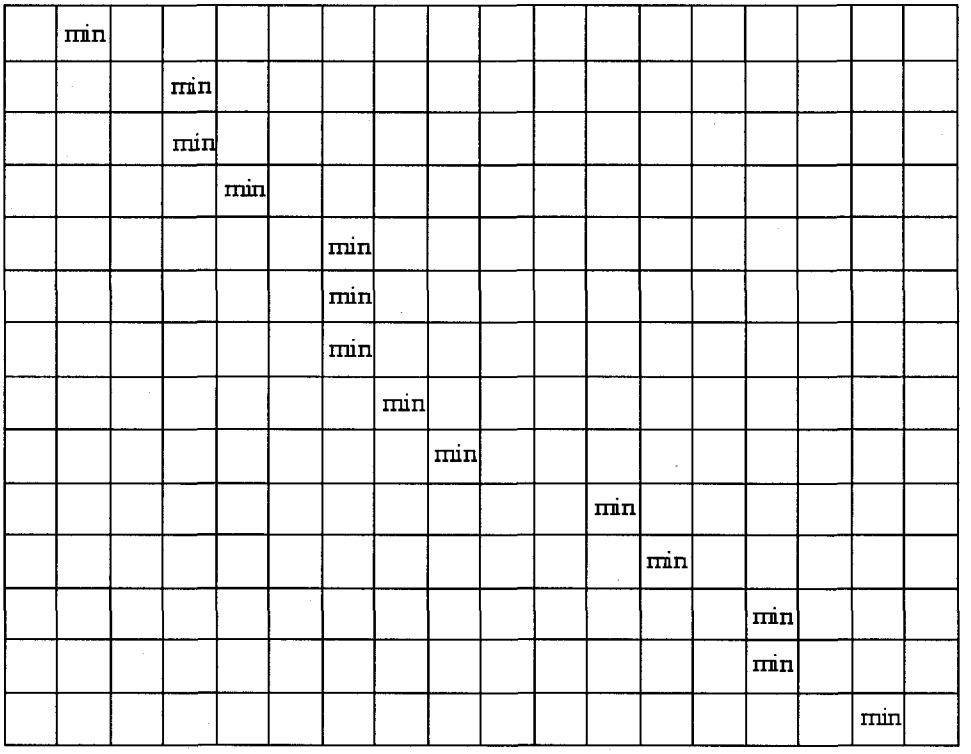


Figure 4: Minima in Totally Monotone Matrix.

As a result in a totally monotone matrix row minima veer to the right as one travels down the rows of the matrix (see Figure 4.) Below is a totally monotone matrix. The row minima are shown in bold. Note that the underlined elements show that this is not a Monge matrix

12 66 61 75 91 75 83 93 91 98 97 98 99

74 85 84 91 94 88 89 92 90 96 95 96 97

16 48 20 85 89 43 43 56 43 89 84 57 90

14 37 32 92 93 24 24 33 24 91 33 32 88

72 81 70 84 85 69 61 70 53 72 66 **52** 70

29 22 19 21 20 18 16 17 14 15 13 **11** 12

2.4 Usefulness of Monge Property for Solving Computer Science Problems:

How is Monge property useful in solving Computer Science problems?

Many of the computer science problems in real life have similar properties or close to sub-properties of Monge arrays. As I will describe next, there are a number of applications where Monge properties play an important role. Some of these depend on the fact that an underlying Monge property makes a greedy approach work. Other applications rely on fast matrix searching. One such problem will be locating minimum entry in a row. If one wants to find all row minima of an n×n matrix this will take $O(n^2)$ time if no special structure is known. But if the matrix is totally monotone one can immediately give an O(n log n) using the divide and conquer approach suggested in Figure 5. For this one assumes that any element of the matrix can be queried in O(1) time. This is a reasonable assumption because usually the elements of the matrix are given by some closed form or oracle. A more complex algorithm by Aggarwal et al. [2] can accomplish this in O(n).

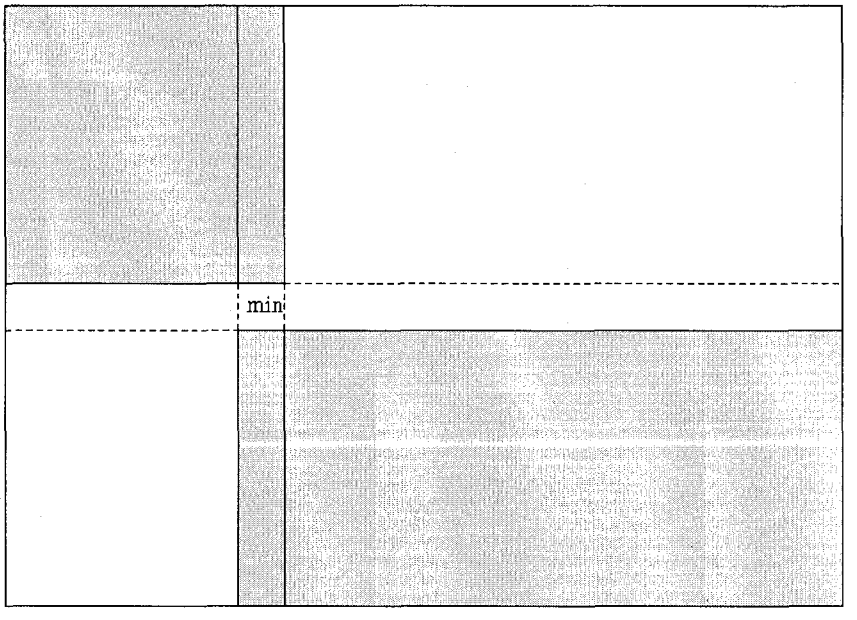


Figure 5: A Divide and Conquer Approach for Row Minima

CHAPTER 3

APPLICATIONS

3.1 Transportation Problem:

One of the first problems considered under a Monge matrix is the transportation problem. The Transportation problem can be explained with this simple example:

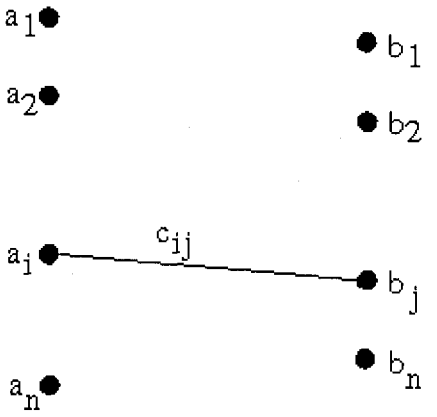


Figure 6: The Transportation Problem

Let the set of points on the left and right represent the ports on two different islands. (See Figure 6.) In the transportation problem one has to ship commodities from the left island to the right. The a values and b values give the number of units of supplies on the left side and the number of units of demands on the right side.The cost to transport one unit from port i to port j is c_{ij}. One assumes that supply matches demand, i.e.

$$\sum a_i = \sum b_j .$$

Then the transportation problem can be written by as

$$\min \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

such that

$$\sum_{j=1}^{m} x_{i,j} = a_i \quad \text{for i=1,...,n}$$

$$\sum_{i=1}^{n} x_{i,j} = b_j \quad \text{for j=1,...,m}$$

Figure 7 illustrates the array x_{ij}; all rows and columns of a solution must add up to the corresponding supply and demand.

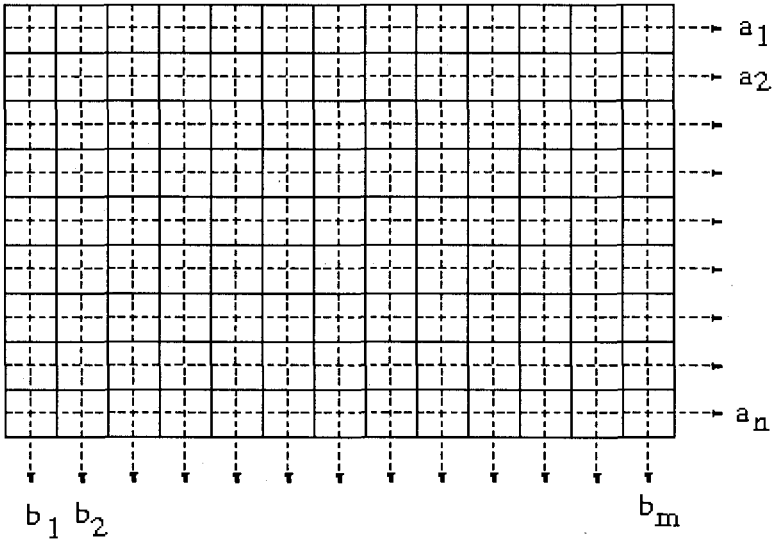


Figure 7: The Solution Matrix for the Transportation Problem

Hoffman showed in his classical result that if the cost matrix is Monge then the optimal solution can be obtained by assigning the x_{ij} values in lexicographical order using the

greedy algorithm. This algorithm fills up (1,1) as much as possible, then (1,2), then (1,3), ..., (1,m) and then (2,1), (2,2) and so forth. This rule is also known as the "North-West corner rule".

Instead of continuing with this example, in the following we will consider the related assignment problem:

3.2 Assignment Problem:

Consider a complete bipartite graph represented by, $G = (V_1 \cup V_2, V_1 \times V_2)$ where $V_1=\{s_1, \ldots\ldots, s_n\}$ and $V_2 = \{t_1, \ldots\ldots t_m\}$ and let c_{ij} be the cost of each mapping of i^{th} element from V_1 to j^{th} element of V_2. In the assignment problem one is to find a one to one mapping which has the minimum cost. A solution of the assignment problem is represented as n×m matrix $X=(x_{i,j})$ over $\{0,1\}$. The presence of a 1 at the intersection of i and j indicates the assigning of the i^{th} node in V_1 to j^{th} node in V_2.

The problem can thus be represented as

$$\sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij}$$

such that

$$\sum_{j=1}^{m} x_{i,j} = 1 \quad \text{for i=1,...,n}$$

$$\sum_{i=1}^{n} x_{i,j} \leq 1 \text{ for j=1,...,m}$$

$x_{i,j} \in \{0,1\}$, i = 1,....n and j = 1,.....,m.

Theorem: Let C be a Monge array of dimension n × m, where $n \leq m$ and $c_{ij} \leq c_{ik}$ when $j \leq k$, then optimal solution for the Assignment Problem will be:

$$x_{ij} = \begin{cases} 1 & \text{when } i = j \\ 0 & \text{otherwise} \end{cases}$$

Proof: Let $y = y_{ij}$ be the optimal solution with $y_{kk} = 1$ for $k = 1,\ldots\ldots,i$, where i is as large as possible. If $i = n$ then we have the solution, now let us assume $i < n$.

Then we have $y_{i+1,i+1} = 0$, thus there exist some index $t > i+1$, such that $y_{i+1,t} = 1$,

Now we have two cases.

Case 1: The (i+1)-st node on the right is taken by some $y_{j,i+1}=1$.

We have $c_{i+1,i+1} + c_{j,t} \leq c_{i+1,t} + c_{j,i+1}$. Thus we can simply switch the two "crossing edges".

Thus if we set

$$\bar{y}_{rs} = \begin{cases} 1 & \text{if } r=s=i+1 \text{ or } r=j, s=t \\ 0 & \text{if } r=i+1, s=t \text{ or } r=j, s=i+1 \\ y_{rs} & \text{else} \end{cases}$$

Then $\bar{y}_{rs}$ is optimal, contradicting the optimality of i.

Case 2: The right node (i+1) is not taken, i.e $y_{k,i+1} = 0$ for all $k \geq i+1$.

There exist a $h > i+1$ with $y_{i+1,h} = 1$. We can get another optimal solution by moving the right end point from h to (i+1). The new solution is:

$$\bar{y}_{rs} = \begin{cases} 1 & \text{if } r=s=i+1 \\ 0 & \text{if } r=i+1, s=h \\ y_{rs} & \text{else} \end{cases}$$

Again, we have a contradiction.

If there are as many nodes on the right as there are on the left then an optimal solution is simply given by

Theorem: Let C be a Monge array of dimension $n \times n$, then optimal solution for the Assignment Problem will be:

$$x_{ij} = \begin{cases} 1 & \text{when } i = j \\ 0 & \text{otherwise} \end{cases}$$

This follows because Case 2 does not occur if n=m.

3.3 Traveling salesman problem (TSP):

Another example where Monge properties make solutions easier is the Traveling salesman problem; the following is from [12].

Traveling salesman problem (TSP): Given n, where $n \in R$, cities and cost matrix $C_{n \times n}$, then the traveling salesman problem is to find a tour that starts and ends at a city i , where $1 \leq i \leq n$, passing through all the cities exactly once, and has a minimum overall cost. The following notation gives the mathematical representation of TSP:

Find a permutation such that

$(\sum_{i=1}^{n} C_{i,t(i)} + C_{n,1})$ is minimized

(t(i) denotes the successor of i in the permutation.)

In general, TSP is NP-hard. But there is a dynamic programming solution for shortest pyramidal tour in $O(n^2)$ time if the underlying cost matrix is Monge.

Definition: A tour t is said to be pyramidal if, t is of the form 1, i1, i_2, i_3, ..., n, j_1, ... , j_{n-r-2}), where 1<i_1<i_2<i_3<.... and j_1>j_2>j_3>...>j_{n-r-2}.

Definition: A peak in a tour lambda is a city j for which both the preceding $t^{-1}(j)$ and succeeding t(j) city's labels are less than j and a valley is where both the preceding and succeeding city labels are greater than j.

Theorem: If T is a Monge Matrix then there exists an optimal tour which is pyramidal

Proof: Proof by induction on the tour of n cities.

Consider a tour of 2 cities, this is inherently pyramidal as the only possible tours are (1, 2) or (2, 1).

We assume that the tour on n-1 cities is pyramidal.

Now let the tour on n cities is not pyramidal, implies that the tour T has some peaks (We also assume that label of city at peaks is not equal to n). Let 'j' be a peak and let i be the location right before j and k be the location right after j. We can replace the segments i, j, k by i, k resulting a sub-tour, which is no longer than the original tour. This is because we have

$T[i,k] + T[j,j] \leq T[i,j] + T[j,k]$

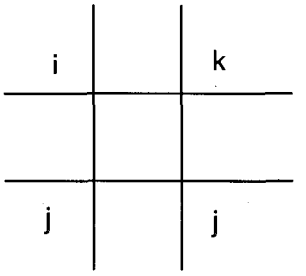


Figure 8: Different Positions

$(C[i\_1,j\_1] + C[i\_2,i\_2] \leq C[i\_1,i\_2] + C[i\_2,j\_1]$ applying Monge property at the peak)

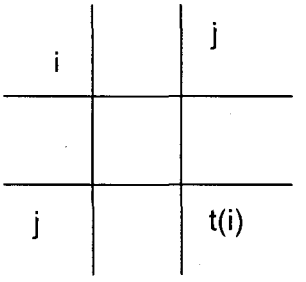


Figure 9: Different Positions

Now we have two sub tours one having city j only and the other t' having n-1 cities. Now let us insert the city j into the tour t' such that $i<j<t(i)$, $t^{-1}(j) = i$ and $t(j)= t(i)$ resulting in a tour t'' which is pyramidal, the overall cost of the tour will be no greater than the initial tour.

We now give a dynamic program to find a pyramidal tour.

Let H[i, j] be the shortest Hamiltonian path from i to j on cities {1,2,3...max{i, j}}, satisfying the condition that the Hamiltonian path passes from i to 1 in descending order and through the remaining cities in ascending order i.e., from 1 to j. We can indicate the above scheme as follows:

$$H(i,j) = \begin{cases} H(i, j-1) + c_{j-1,j} \text{ for } i < j-1, \\ \min\limits_{k<i} \{H(i, k) + c_{kj}\} \text{ for } i = j-1, \\ H(i-1, j) + c_{i,i-1} \text{ for } i > j+1, \\ \min\limits_{k<j} \{H(k, j) + c_{ik}\} \text{ for } i = j+1. \end{cases}$$

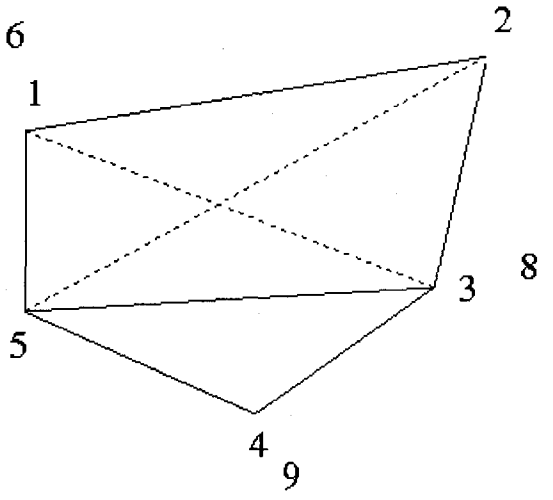
The length of a shortest pyramidal tour is given by:

$$\min\{H[n-1,n] + t_{n,n-1} , H[n,n-1] + c_{n-1,n}\}.$$

The dynamic program can be implemented in $O(n^2)$.

3.4 All Farthest Points:

The next application is from computational geometry. The problem is to find all farthest neighbors in a polygon. When we take the clockwise distances of the polygon and insert these values into a matrix as in Figure 10 then the matrix is reverse Monge. Thus all maxima can be computed in linear time.



$$C = \begin{pmatrix} 0 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -2 & -1 & 0 & -1 & -1 \\ -3 & -2 & -1 & 0 & -1 \\ -4 & -3 & -2 & -1 & 0 \end{pmatrix}$$

CLOCK WISE DISTAN CES

Figure 10: Farthest Points

CHAPTER 4

BATCHING PROBLEMS

The main chapter of this thesis deals with batching problems from the theory of scheduling. The outline of the chapter is as follows. First, I consider the s-list batch problem. Here the job order is given, and the algorithm has to decide how to batch. The algorithm can be made to run in linear time because of a Monge-like property. Indeed, the property is stronger than the Monge property, which makes it even easier to come up with a linear time algorithm. It is crucial that the order of jobs is given. The problem where the algorithm has to find the best order is NP-hard. This problem is called s-batch problem. As a heuristic for the s-batch problem I will describe an approach based on simulated annealing: For any given order the best batching can be found in linear time. The simulated annealing method searches through many orders. Eventually it finds an order with a solution close to optimal. For each order searched by simulated annealing the linear s-list batch algorithm is used as a subroutine.

First I describe what a batching problem is:

4.1 Batching Problem:

Grouping a set of jobs into batches and scheduled on a single machine is a batching problem. Jobs belonging to same batch are processed jointly. The completion time of each job is the completion time of the last job batch. Each batch needs a set-up

time before each batch is scheduled, which is assumed to be the same for all batches. Since the jobs are processed sequentially, the problem we are considering is denoted as s-batch problem. Consider a set of jobs J = $\{j_i\}$ with processing times P = $\{p_i\}$ and corresponding weights W = $\{w_i\}$, where i = {1,…., n} to be arranged into batches B = $\{b_j\}$ where j={1,…,m} with a minimum amount of total processing time. The Objective function for s-Batch problem is the overall completion time denoted by $\sum w_i C_i$, where C_i denotes the completion time of j_i in a particular schedule. In order to find the optimum scheduled we have to look for a schedule that has a minimum value of the objective function.

Consider the following example, a 6-job problem where processing times are p_1=1, p_2= 3, p_3=5, p_4=7, p_5=5, p_6=4, and weights are w_1=1, w_2 =2, w_3=3, w_4 =3, w_5=2, w_6=1.

The batching problem we are considering schedules jobs sequentially, thus we refer to the batching problem more specifically as s-batch problem.

S	J_1	J_2	S	J_3	S	J_4	S	J_5	S	J_6

$\sum w_i C_i = 185$

S	J_2	J_3	S	J_4	J_1	S	J_5	S	J_6

$\sum w_i C_i = 194$

S	J_3	J_2	S	J_4	S	J_5	S	J_6	J_1

$\sum w_i C_i = 200$

S	J_5	S	J_4	S	J_3	S	J_6	J_2	S	J_1

$\sum w_i C_i = 242$

As we can observe the value of the objective function varies with different batches. Batching of jobs ordered according some order of precedence is known as the s-list batch problem. Often the precedence criteria is taken as priority which is the value of $\square_i = \frac{W_i}{P_i}$ where W_i and P_i are the weight and processing time of a job J_i respectively. But this order does not always give the optimal solution.

I now turn to the problem where the order of the jobs is given, i.e. the s-list batch problem. In this case the problem can be reduced to a special case of shortest path problem:

Consider an arbitrary job sequence $S = J_1J_2J_3J_4.....J_n$ for a given batching problem, the goal is to minimize the overall processing time represented by the relation

$$E = \sum \alpha_i f_i$$

Solution for an s-batch problem will be of the form

$$B = J_0\, SJ_{i1}.....J_{i2-1}SJ_{i2}.....J_{i3-1}S.....SJ_k.....J_n$$

Where k is the number of batches and n is the number of jobs.

The completion time of each batch is given by

$$Pj = s + \sum_{k=i_j}^{i_{j+1}-1} p_k$$

The relation for the shortest path can be represented for the batching problem as

$$E(B) = \sum_{i=1}^{n} \alpha_i f_i$$

The above equation is equal to the processing sum of the processing times of all the batches,

$$E(B) = \sum_{j=1}^{k} \left(\sum_{v=i_j}^{n} \alpha_v \right) \left(s + \sum_{v=i_j}^{i_{j+1}-1} p_v \right)$$

In order to solve the batching problem, we have to find k and a series of indices i_1, i_2, i_3,....., i_n, so as to minimize the objective function. This problem can be reduced to shortest path problem of the form:

$$B = J_0\, SJ_{i1}.....J_{i2-1}SJ_{i2}.....J_{i3-1}S.....SJ_k.....J_n$$

each batch represents a path, whose cost can be given by the relation

$$C_{ij} = (W_n - W_i)(s + P_j - P_i)$$

C_{ij} is the cost of the path between Ji and the batch containing $J_{i+1}.....J_{j-1}$

Solution for a list batch problem can be reduced to that of a shortest path algorithm, which has the following cost equation for a path between two jobs i and j.

$$C_{ij} = (W_n - W_i)(s + P_j - P_i)$$

where $W_n = \sum_{i=0}^{n} w_i$ $W_i = \sum_{k=0}^{i} w_k$,

s is the set up time,

$$P_j = \sum_{k=0}^{j} p_k \qquad P_i = \sum_{k=0}^{i} p_k$$

C_{ij} are the costs of the paths between the first job of each batch J_i, J_{i+1},....., J_{k-1}.

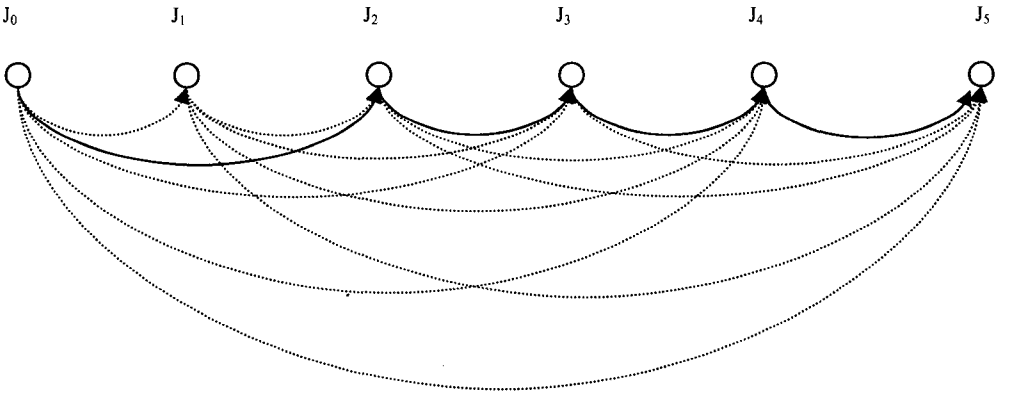


Figure 11: Reduction of Batching Problem to a Shortest Path Problem

In order to find the optimum schedule we just have to find the minimum path from 1 to n, in which a new batch is formed at each node on the path. For example, if we have a minimum cost path as 0-2-3-4-5, then the jobs will be batched as following:

S	J_1	J_2	S	J_3	S	J_4	S	J_5

Let E[k] represents the minimum cost of the path from node 0 to 'k', then objective function for the S-Batch problem would be to find minimum of E[n].

$$E[k] = \begin{cases} \min\{E[i] + Ci_k\}, & \text{when } k \neq 0 \\ 0, & \text{when } k = 0 \end{cases}$$

where C_{ik} is the cost of the path from i to k.

$E[0]+C_{01}$					
$E[0]+C_{02}$	$E[1]+C_{12}$				
$E[0]+C_{03}$	$E[1]+C_{23}$	$E[2]+C_{23}$			
$E[0]+C_{04}$	$E[1]+C_{14}$	$E[2]+C_{24}$	$E[3]+C_{34}$		
.....					
.....					
.....					
$E[0]+C_{0n}$	$E[1]+C_{1n}$	$E[2]+C_{2n}$	$E[3]+C_{3n}$	.....	$E[n-1]+C_{n-1n}$

As we can see that the cost matrix is the lower half of a n×n matrix, and to find the minimum cost path in each row and to find E[n] has a worst case of $O(n^2)$. In order to improve the algorithm we use the inherent Monge property of the cost matrix.

4.2 Product Property:

In fact the matrix has a stronger property. The cost matrix also represents a special case of shortest path problem which is:

$C[i, l] - C[i, k] = f(i)\, h(k,l)$, where $h(k,l) \geq 0 \ \forall \ i<k<l$

where f(i) is a non-increasing function.

The mentioned property is called the *Product property*. Following is the proof that the above property satisfies the Monge property:

We know, the Monge property is

$$C[i,j] + C[k,l] \leq C[i,l] + C[k,j]$$

Now by rearranging the terms we get,

$$C[i, j] - C[i, l] \leq C[k, j] - C[k, l]$$

By applying product property we get,

$$f(i)\, h(l, j) \leq f(k)h(l, j)$$

By canceling out the common terms we get,

$$f(i) \leq f(k) \text{ which is given.}$$

Hence the C is a Monge matrix.

Our matrix

$$C_{ij} = (W_n - W_i)(s + P_j - P_i)$$

has the product property:

$$C[i, l] - C[i, k] = (W_n - W_i)(s + P_l - P_i) - (W_n - W_i)(s + P_k - P_i)$$

$$= (W_n - W_i)(P_l - P_k) = f(i)\, h(k,l)$$

with

$$f(i) = (W_n - W_i) \text{ non-increasing,}$$

and

$h(k,l) = (P_l - P_k) \geq 0.$

We now describe an algorithm which exploits this property and is linear time.

The algorithm is taken from [3] and [9].

Let

F(i) be the length of a shortest path from i to n for i≤n.

Set

$$F(i,k) = c_{i,k} + F(k) \quad for \quad i < k.$$

Furthermore define

$$V(k,l) = \frac{F(k) - F(l)}{h(k,l)} \quad k<l.$$

Without proof we note:

Fact 1: Assume that $\mathcal{V}(k,l) \leq f(i)$ for some $1 \leq i < k < l \leq n$. Then $F(j,k) \leq F(j,l)$ holds for all $j=1,\ldots,i$.

Fact 2: Assume that $\mathcal{V}(i,k) \leq \mathcal{V}(k,l)$ for some $1 \leq i < k < l \leq n$. Then for each $j=1,\ldots,i$ we have $F(j,i) \leq F(j,k)$ or $F(j,l) \leq F(j,k)$.

The proof of the two facts is in [9].

4.3 Queue-Process Algorithm:

Then the following algorithm "Queue-Process" finds the shortest path in linear time [3,8].

Algorithm *Queue-Process*

1. $Q := \{n\}$; $F(n) := 0$;
2. FOR $i := n$-1 DOWN TO 1 DO

   BEGIN
3. WHILE head(Q) $\neq$ tail(Q) and $f(i) \geq \nu$ (next(head(Q)), head(Q)) DO

   delete head(Q) from the queue;
4. $SUCC\ (i)$:= head(Q);
5. $j := SUCC\ (i)$;
6. $F\ (i) := c_{ij} + F\ (j)$;
7. WHILE head(Q) $\neq$ tail(Q) and $\nu\ (i,$ tail(Q)) $\leq$ ν (tail(Q), previous (tail(Q))) DO

   delete tail(Q) from the queue;
8. Add i to the queue

   END

In the next chapter I describe my implementation of this algorithm. The program is very fast. If the order of the jobs is given, then the optimal batching can be generated fast even for a very large number of jobs.

However, if we also are allowed to change the job order, then the problem is NP-hard. We use simulated annealing to search the space of all orders; for each specific order we use the program Queue-Process as a subroutine.

4.4 Simulated Annealing:

I will now describe simulated annealing (SA). SA is inspired by annealing in metallurgy, a technique involving cooling of a material to increase the stability of formed metal. One starts with some permutation as the current solution (cur). Then one generates a neighbor: A neighbor is any permutation that can be obtained by swapping two elements of the permutation. If the new permutation (new) has a lower value, then we accept that solution as the new current solution, but if it is worse (larger) it is not always rejected. It is accepted with probability $P_{c_k}\{accept \quad new\}$ given below. The probability P_{c_k} is depended on the "temperature" c_k. Initially, a high value means that worse solutions are accepted with relatively large probability. As the simulation progresses the temperature is lowered and acceptance probability for worse solutions is smaller.

$$P_{c_k}\{accept \quad new\} = \begin{cases} 1 & if \quad f(new) \le f(cur) \\ \exp(\dfrac{f(cur) - f(new)}{c_k} & if \quad f(new) > f(cur) \end{cases}$$

The different values of k are called the phases of the simulation.

CHAPTER 5

COMPUTER IMPLEMENTATION

5.1 Variables and Data Used in the Code:

The objective of the code is to find a minimum schedule by continuously improving the Objective function. The user has only to enter the number of jobs and the required data i.e. the processing times and weights are produced using myrand() function. We are generating random processing times in the range (.001 – 1.5) and random weights in the range of (1-9). The initial order in which the jobs are to be processed/scheduled is taken as the sequential order from (1-n), and the value of the Objective function as a result of the scheduling the jobs in the initial order will be initial minimum. As was discussed earlier we use "Simulated Annealing" to arrive at a better solution. In order to process the jobs using simulated annealing we will need a constant that is reduced by a factor after each step. We generate the Constant (Temperature in Simulated Annealing of metals) by taking the square root of the number of jobs, this constant is reduced to 70% percent at each stage. The number of trials is set to 200 for each constant, at each iteration we swap elements of the two randomly generated indexes and jobs are processed/ scheduled in the newly generated order. The cost of the path for the current schedule is compared to the previous value and is accepted if it is found to improve the Objective function. Once the jobs are scheduled, the order is changed to the one which has better schedule. If the new schedule does not improve the objective function, new

schedule is discarded and we go back to the schedule which has previous best. The exception for the above rule is that we sometimes accept the worse value based on the following relation.

Prob = 100 * e(Current_minimum _ value-Worse_value/C) > myrand(1,100)

By accepting the worse values sometimes we are avoiding the risk of getting trapped in the local minima, in order to find the global minima, which is the optimum value for the Objective function in our case. This process is continued till we reach a threshold value for the constant which we have set as 10% of the initial value for the constant.

Integer pointer pw: Stores the processing times and corresponding weight for the given jobs.

Integer pointer PS: Stores the partial sums of processing times and weights.

For example PS[i][1] = $\sum_{j=0}^{i} pw[j][1]$ and PS[i][2] = $\sum_{j=0}^{i} pw[j][2]$

Integer pointer PERM: Stores the order in which jobs have to be considered.

NUM_TRIAL: Number of trials for each constant.

C: We take square root of the number of jobs to be scheduled and this value is reduced to 70% once we repeat the process of finding the minimum schedule for set number of times.

Float pointer F: Stores the minimum value in each row and the column number in which the minimum is found.

Character pointer LIST: Stores the T for (True) and F for (false), a T in ith position indicates that the job at that position has a setup time before that and a F indicates that it does not have a set up time before it.

Structure indexq: track is used to maintain a circular queue to calculate the minimum of each row.

5.2 Functions Used:

Function ij_to_k(int , int, int): Function ij_to_k() is used to dynamically access the elements of an array. Function ij_to_k() takes in three integer arguments i, j and n, where i and j give the position of an element in the array and n is the number of columns in the array. It returns an integer index which gives the position of the current element an array if stored in a single dimension.

Function myrand(int, int): mrand() generates a random integer between the two integer arguments passed to it.

Function costij(int, int, int): costij() calculates the cost of the path from i to j. We assume setup time s=1.

Function compf(): compf() calculates the value of the equation f(n) = W[n]-W[i], where n is the number of jobs , W[n] is the sum of weights of n jobs and W[i] is the sum of the weights of jobs from 1 to i.

Function compv(): compv() calculates the value of the equation v=(F(k)-F(l))/(P[l]-P[k]), where F(k) and F(l) are the minimum values in the kth and lth row respectively and P[l] and P[k] are the partial sums of the processing times, i.e, P[k]= P[n]-P[k].

Function add(int, int): add() adds elements to the circular queue. A queue is used as a support to find minimum value in a row. It takes two integer arguments one is the column number to be added and the other number of jobs.

Function newperm(int, int): newperm() swaps two elements of an array.

Function altlist(): altlist() calculates the minimum value in each row. In order to do this it starts with a for loop from n (number of jobs) until 0. The first while loop deletes head of the queue until it satisfies the condition: head(Q)!=tail(Q) and f(i)>=v(next(head(Q)), head(Q)). Second while loop deletes the tail of the queue if it satisfies the condition: head(Q)!=tail(Q) and v(i, tail(Q)) <= v(tail(Q), previous(tail(Q))

Function Quick_sort(int,int): Quick_sort() is used to sort the processes in a priority order which is non-increasing order of w_i/p_i, where w_i is the weight and p_i is the processing time.

Function partition(int,int,int): partiotion() creates a partition by taking the first element in the structure array p_w, and finding the right position in the array.

Function main(): On running the following code it asks for the number of jobs to be read and rest of the data for processing time and corresponding weights will be generated automatically. Initial temperature is set to the square root of the number jobs given and number of trials for each reduction of temperature is set as a constant value for 200. Initial order of the jobs is set as (1, 2, 3,.... , n) in that order. So initial batch with that order is calculated, the initial minimum and batch are set. Now, while the initial Constant value C is greater than threshold we look improve on the initial minimum value obtained.

The way we improve on the solution is as follows: We take the initial order, generate two random indices and swap elements at these places to generate a new order for the jobs.

The jobs are then processed in the new order and a new minimum value and a new batch is calculated. The new value calculated is then compared to the current minimum value and

5.3 Code:

```
#include<stdio.h>
#include<iostream.h>
#include<stdlib.h>
#include<math.h>

//Number of trials for each constant
const int NUM_TRIAL=200;
//integer pointer for the order of jobs
int *PERM;
//float pointer for processing times, weights and partial sums
float *pw, *PS;
int *E_min, *work;
//float pointer to the moge array
float *F;
//Character array for the final schedule
char *LIST;
//structure to calculate the minimum in each row
struct indexq{
        int *col;
        int head, rear, n;
}track;
struct processtime_weight{
```

```
float time;
float weight;
float priority;
};
processtime_weight *pw;
//function ij_to_k () returns the location of the current element in the given array
int  ij_to_k(int i, int j, int n)
{
 return i*n+j;
}
//function myrand() returns a random number between lower and upper integers sent
int myrand(int lower, int upper)
{
        return (lower+rand()%(upper-lower+1));
}
// function  costij() computes cost of path from i to j
float costij(int k, int l,  int n)
{
int s=1;
return((PS[ij_to_k(n, 2, 2)]-PS[ij_to_k(k, 2, 2)])*(s+PS[ij_to_k(l,1, 2)]-PS[ij_to_k(k, 1,
2)]));
}
// function compf() computes the value of f(i) = (W[n]-W[i])
```

```
float compf(int i, int n)
{
return ((PS[ij_to_k(n, 2, 2)] - PS[ij_to_k(i, 2, 2)]));
}
// function compv() computes the value of v=(F(k)-F(l))/(P[l]-P[k])
float compv( int prevhead, int head)
{
        return ( ( (F[ij_to_k(prevhead, 1, 2)] - F[ij_to_k(head, 1, 2)])/(PS[ij_to_k(head, 1,
2)]-PS[ij_to_k(prevhead, 1, 2)])));
}
//function add() adds a new element to the queue
void add(int i, int n)
{
int t;
t = (track.rear+1)%n;
if(t == track.head)
cout<<"\nQueue Overflow\n";
else
{
track.rear=t;
track.col[track.rear]=i;
}
}
```

```
/*Function for partitioning the array*/
int Partition(int low,int high)
{
int i;
float
high_vac=0,low_vac=0,pivot=0,pivot_time=0,pivot_weight=0,high_time=0,high_weight
=0,low_time=0,low_weight=0;
  pivot=pw[low].priority;
  pivot_time = pw[low].time;
  pivot_weight = pw[low].weight;
  while(high>low)
        {
        high_vac=pw[high].priority;
        high_time =pw[high].time;
        high_weight = pw[high].weight;
 while(pivot>=high_vac)
 {
  if(high<=low) break;
  high--;
  high_vac=pw[high].priority;
  high_time=pw[high].time;
  high_weight = pw[high].weight;
 }
```

```
            pw[low].time=high_time;
            pw[low].weight=high_weight;
            pw[low].priority=high_vac;
            low_vac=pw[low].priority;
            low_time = pw[low].time;
            low_weight =pw[low].weight;
  while(pivot<low_vac)
  {
   if(high<=low) break;
   low++;
   low_vac=pw[low].priority;
   low_time = pw[low].time;
   low_weight = pw[low].weight;
  }
  pw[high].time=low_time;
  pw[high].weight = low_weight;
  pw[high].priority = low_vac;
}
  pw[low].priority=pivot;
  pw[low].time = pivot_time;
  pw[low].weight = pivot_weight;
 return low;
}
```

```
void Quick_sort(int low,int high)
{
 int Piv_index,i;
 if(low<high)
 {
  Piv_index=Partition(low,high);
  Quick_sort(low,Piv_index-1);
  Quick_sort(Piv_index+1,high);
 }
}
//function altlist() calculates the minimum in each row of the array
void altlist(int n){
int i;
track.n=n;
track.head=track.rear=0;
PS = new float[n*2+n];
F  = new float[n*2+n];
PS[ij_to_k(0, 1, 2)]= 0;
PS[ij_to_k(0, 2,2)]=0;
track.col = new int[n+1];
        for(i=1 ;i<=n ; i++)//calculating partial sums of weights and processing times
        {
                PS[ij_to_k(i, 1, 2)]= PS[ij_to_k(i-1, 1, 2)]+
```

```
                    pw[ij_to_k(PERM[ij_to_k(i,1,1)], 1, 2)];
            PS[ij_to_k(i, 2, 2)] = PS[ij_to_k(i-1, 2, 2)]+
                    pw[ij_to_k(PERM[ij_to_k(i,1,1)], 2, 2)];
    }
    track.col[track.head]=track.col[track.rear]=n;
    F[ij_to_k(n, 1, 2)]=0; F[ij_to_k(n, 2, 2)]=0;
    //code to find the shortest path
    //First while loop deletes head of the queue until it satisfies the condition:
head(Q)!=tail(Q) and f(i)>=v(next(head(Q)), head(Q))
    //Second while loop deletes the tail of the queue if it satisfies the condition:
head(Q)!=tail(Q) and v(i, tail(Q)) <= v(tail(Q), previous(tail(Q))
    for( i = n-1; i>=0; i--)
    {
        int temp= track.head+1;
        while (track.col[track.head] != track.col[track.rear] && compf(i, n) >=
                compv( track.col[temp], track.col[track.head]))
        track.head = (track.head + 1)%n;
        int j = track.col[track.head];
        F[ij_to_k(i, 1, 2)] = costij(i, j, n) + F[ij_to_k(j, 1, 2)];
        F[ij_to_k(i, 2, 2)] = j;
        int temp2 = track.rear-1;
        while(track.col[track.head] != track.col[track.rear] &&
            compv(i, track.col[track.rear]) <=
```

```
                    compv(track.col[track.rear], track.col[temp2]))
            track.rear = (track.rear-1)%n;
            add(i, n);
        }
}
//function batch() batches the jobs based on the minimum array calculated
void batch(int n){
int i;
float temp;
LIST = new char[n+10];
i=1;
temp = F[ij_to_k(0,2,2)];
LIST[ij_to_k(0,1,1)]='T';
        while(i<n)
        {
            if(temp!=F[ij_to_k(i,2,2)])
            {
                LIST[ij_to_k(i,1,1)]='T';
                temp=F[ij_to_k(i,2,2)];
            }else
            {
                LIST[ij_to_k(i,1,1)]='F';
            }
```

```
        i++;
        }
}

//function newperm() swaps two elements in the PERM[] to generate a new order
void newperm(int index_i,int index_j){
int temp = PERM[index_i];
PERM[index_i]=PERM[index_j];
PERM[index_j]=temp;
}
int main(){
float C;
char another_order;
int n,i=0,j=0,index_i=0,index_j=0;
int no_of_trials=0;
float cur_value;
char *cur_batch;
int *cur_perm;
float threshold;
        cout<<"\n Enter number of jobs";
        cin>>n;
        no_of_trials = NUM_TRIAL;
        C = sqrt(n);
```

```
threshold=.1*C;
cur_batch = new char[n+10];
cur_perm = new int[n+10];
pw = new float[n*2+n];
PERM = new int[n+10];
srand(time(NULL));
pw[ij_to_k(0, 1, 2)]=0; pw[ij_to_k(0, 2, 2)]=0;
for(i=1; i<=n ;i++)
{
        pw[ij_to_k(i, 1, 2)]= myrand(1, 1500)/1000;
        pw[ij_to_k(i,2,2)]=myrand(1,9);
}
PERM[ij_to_k(0,1,1)]=0;
for(i=1;i<=n;i++)
PERM[ij_to_k(i,1,1)] = i;//intial order of the jobs
altlist(n);
batch(n);
cur_value = F[ij_to_k(0,1,2)];
cout<<"\n Initial minimum value is "<<cur_value<<"\t Initial list is: ";
cur_perm[ij_to_k(0,1,1)]=0;
for(i=0;i<n;i++){
cout<<LIST[ij_to_k(i,1,1)];
cur_perm[ij_to_k(i+1,1,1)] = PERM[ij_to_k(i+1,1,1)];
```

```
}
delete track.col;
delete PS;
delete F;
delete LIST;
//Repeat the process till the constant reaches a threshold value
while(C > threshold){
cout<<"**********The current value of the constant is:
        "<<C<<"**********"<<endl;
        for(i=0;i<no_of_trials;i++)
        {
                index_i=myrand(2,n);
                do{
                        index_j=myrand(2,n);
                }while(index_j==index_i);
newperm(index_i,index_j);
altlist(n);
batch(n);
float outcome1 = 100 *  exp(((cur_value-F[ij_to_k(0,1,2)]))/C);
float outcome2 = prob(1,100);
// cout << endl;
// cout << "here" << endl;
//cout <<  100 *  exp(((cur_value-F[ij_to_k(0,1,2)]))/C) << endl;
```

```
// cout << "here" << endl;
if(F[ij_to_k(0,1,2)]<=cur_value){
        cout << endl;
        cout <<"IMPROVED SOLUTION----------------------------------------------
                --"<< endl;
        cout<<"\nThe new minimum value is and the order and batch is\n";
        cur_value = F[ij_to_k(0,1,2)];
        cout<<cur_value<<"\n";
        for(j=1;j<=n;j++)
        {
                cur_perm[ij_to_k(j,1,1)]=PERM[ij_to_k(j,1,1)];
                cur_batch[ij_to_k(j-1,1,1)] = LIST[ij_to_k(j-1,1,1)];
        }
        for(j=1;j<=n;j++)
        {
                cout<<cur_perm[ij_to_k(j,1,1)];
        }
        cout<<"\n";
        for(j=0;j<n;j++){
                cout<<cur_batch[ij_to_k(j,1,1)];
        }
        }else if (outcome2 < outcome1)
        {
```

```
                //*********************************

                cout << endl;

                cout  <<"WORSE  ACCEPTED----------------------------------------------
"<< endl;

      cout << "Temperature: " << C << endl;

                cout<<"\nThe new (worse) value is and the order and batch is\n";

                cur_value = F[ij_to_k(0,1,2)];

                <<cur_value<<"\n";

                for(j=1;j<=n;j++)

                {

                        cur_perm[ij_to_k(j,1,1)]=PERM[ij_to_k(j,1,1)];

                        cur_batch[ij_to_k(j-1,1,1)] = LIST[ij_to_k(j-1,1,1)];

                }

                for(j=1;j<=n;j++){

                        cout<<cur_perm[ij_to_k(j,1,1)];

}

                cout<<"\n";

                for(j=0;j<n;j++){

                        cout<<cur_batch[ij_to_k(j,1,1)];

                }

        }else{

                for(j=1;j<=n;j++)

                PERM[ij_to_k(j,1,1)]=cur_perm[ij_to_k(j,1,1)];
```

```
            }
            delete track.col;
            delete PS;
            delete F;
            delete LIST;
        }
C=.7*C; }}
```

BIBLIOGRAPHY

[1] E. Aarts, J.K. Lenstra, Local Search in Combinatorial Optimization, Wiley, 1997.

[2] A. Aggarwal, M. Klawe, S. Moran, P. Shor and R. Wilber, Geometric applications of a matrix searching algorithm, Algorithmica 2, pp. 195-208 (1987).

[3] S. Albers, P. Brucker, The complexity of one-machine batching problems. Discrete Applied Mathematics 47, 87-107, (1993).

[4] W. Bein, P. K. Pathak. A Characterization of the Monge Property and its Connection to Statistics, Demonstration Mathematica, Vol. XXIX, No. 2, 451-457 (1996).

[5] W. Bein, P. Brucker, L. L. Larmore., J. K. Park. The Algebraic Monge Property and Path Problems. Discrete Applied Mathematics, 145(3): 455-464 (2005).

[6] W. Bein, P. Brucker, L. L. Larmore., J. K. Park. Fast Algorithms with Algebraic Monge Properties. In Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science (MFCS 2002) Warsaw, Poland, August 2002,

[7] W. Bein, P. Brucker, A. J. Hoffman. Series parallel composition of greedy linear programming problems. Mathematical Programming, 62: 1-14 (1993). Reprinted in: A. J. Hoffman (Author), Charles A. Micchelli (Editor), Selected Papers of Alan Hoffman: With Commentary. World Scientific Publishing Company, 353 - 366, (2003)

[8] W. Bein, P. Brucker, P. K.Park, P.K Pathak. A Monge Property for the d-Dimensional Transportation Problem. Discrete Applied Mathematics, 58: 97-109 Lecture Notes in Computer Science, Volume 2420, Springer Verlag (2002), 104-117.

[9] P. Brucker, Scheduing, 5th Edition, Springer Verlag, 2007.

[10] R.E. Burkhard, B Klinz, R. Rudolf, Perspectives of Monge properties in optimization, Discrete Applied Mathematics 70, 95-161 (1996).

[11] A. J. Hoffman, On simple linear programming problems, Proc. Symposia in Pure Mathematics VII, 317-327, Amer. Math. Soc. (1963). Reprinted in: A. J. Hoffman (Author), Charles A. Micchelli (Editor), Selected Papers of Alan Hoffman: With Commentary. World Scientific Publishing Company, (2003)

[12] E. L. Lawler, J. K. Lenstra, A. H. G. Rnnooy Kan, D. B. Shmoys, The Traverling Salesman Problem, Wiley, 1985.

[13] J.K. Park, The Monge array: an abstraction and its application, Ph.D. Thesis, MIT, Masachusetts, 1991

VITA

Graduate College
University of Nevada, Las Vegas

Revanth Pamballa

Address:
4224, Cottage Circle, Apt # 4
Las Vegas, NV 89119, USA.

Degree:
Bachelor of Technology, Computer Science and Information Technology, 2005
Jawaharlal Nehru Technological University, Hyderabad, India

Thesis Title: "A Study of Monge Matrices with Applications to Scheduling"

Thesis Examination Committee:
Committee Chairperson, Dr. Wolfgang Bein, Ph. D.
Committee Member, Dr. Ajoy K. Datta, Ph. D.
Committee Member, Dr. Lawrence L. Larmore, Ph. D.
Graduate College Representative, Dr. Vekantesan Muthukumar, Ph. D.