Graph Partitioning: A Heuristic Procedure to Partition Network Graphs

Thesis submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology

in

Computer Science and Engineering

by:

Deepak Mishra (10506010)

Pramit Das (10506017)

Under guidance of

Prof. A. K. Turuk

Department of Computer Science and Engineering
National Institute of Technology
ROURKELA

National Institute of Technology
Rourkela

CERTIFICATE

This is to certify that the work in this Thesis Report entitled **"Graph Partitioning: A Heuristic Procedure to Partition Network Graphs"** submitted by **Deepak Mishra** and **Pramit Das**, has been carried out under my supervision and guidance, in partial fulfillment of the requirements for the degree of **Bachelor of Technology** in Computer Science during session 2005-2009 in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela.

To the best of my knowledge, the matter embodied in the thesis is authentic and has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Dated: 11th May, 2009
Place: NIT, Rourkela

Prof. A. K. Turuk

CONTENTS

ACKNOWLEDGEMENTS

No thesis is created entirely by an individual, many people have helped to create this thesis and each of their contribution has been valuable. We express our sincere gratitude to our thesis supervisor, Prof. A. K. Turuk, Department of Computer Science and Engineering, for his kind and able guidance for the completion of the thesis work. His consistent support and intellectual guidance made us energize and innovate new ideas.

Last, but not least we would like to thank all the professors and lecturers, and members of the Department of Computer Science and Engineering, National Institute of Technology, Rourkela for their generous help in various ways for the completion of this thesis.

Deepak Mishra (10506010)
Pramit Das (10506017)

B.TECH
COMPUTER SCIENCE & ENGG
2005-2009

ABSTRACT

.

Graphs are mathematical structures used to model pair wise relationship between objects of a certain collection. It consists of collection of vertices or "nodes" and a collection of edges that connect these nodes. Graphs can be directed from one vertex to another or undirected. In our context, a graph denotes a network with computers distributed as nodes while the communication channel acting as the edges. These are directed graphs where each edge has a capacity which cannot be exceeded.

In real life applications, it becomes very essential that graphs are partitioned in some way so as to satisfy certain conditions. For example, while placing components of electronic circuit on circuit boards or substrates, components that are highly dependent on each other (exchanging maximum information) should be placed on the same board. Also an important factor is the number of connections between these boards should be minimized. Similar situation arises in a computer network where computer systems are distributed over a wide geographic location. This is the basis of graph partitioning problem.

The classical graph partitioning problem consists of dividing a graph into pieces, such that the pieces are of about same size and there exists very few

connections between these pieces. The objective is to partition the nodes of a graph with costs on its edges into subsets so as to minimize the sum of the costs on all edges that are cut.

Let G be graph with n nodes, of sizes (weights) $w_i > 0,\ i = 1, 2, ..., n$. Let p be a positive number, such that $0 < w_i < p$ for all i. Let $C = (c_{ij}),\ i,j = 1, 2, ……, n$ be a weighted connectivity matrix describing the edges of G. Let k be a positive integer. A k-way partition of G is a set of disjoint subsets of G, $v_1, v_2, ..., v_k$ such that

A partition is admissible if

for all i. The cost of partition is the
summation of (c_{ij}), where i and j belong to different subsets.

A strictly exhaustive procedure for finding the optimal partition is out of question because the problem of graph partitioning is NP-Hard problem. For a graph with 40 nodes and 4 partitions, the possible number of partitioned cases will be of the order of 10^{36}. Hence, any direct approach to find an optimal solution from these many cases is not a feasible option. As a result heuristic approaches are employed in these cases.

We use a heuristic partitioning algorithm that divides a network into 2 disjoint sets based on the distance between any two nodes. The network used is a real network termed ARPANET and is regarded as the origin of the Internet.

LIST OF FIGURES

INTRODUCTION

1. SCOPE AND MOTIVATION

In this thesis, we consider distance to partition a given network (or graph). Given a graph *G (V, E)*, with edge weight c_{ij} on its edges, we divide the vertices V into 2 sets considering the average distance between each node. That is, the nodes that are closer are kept in the same partition while those that are farther are in the other node. The algorithm can be recursively applied to create further partitions.

The problem being considered is a NP-Hard problem and an exhaustive search method for the optimal partition is infeasible. Hence, we adopt a heuristic method originally proposed by Kernighan and Lin [1].

Because of the wide applications and the difficulty of general graph partition problems, extensive research has been done on general graph partition problems and its variations, including both exact and heuristic algorithms. But surprisingly, we found little literature on partitions considering the physical distance between nodes.

The graph being considered is a real computer network called ARPANET designed by Larry Roberts [2] for U.S. Army in 1960s'. The underlying logic is that the nodes or locations that are geographically proximal should be in the same

partition. This will help better management and more efficient error detection and correction in cases of failures.

The following sections are organized as follows. In the next section, we define various commonly used graph related terminologies. Then we state some graph related problems.

2. BASIC DEFINITIONS

We will first give some definitions from graph theory that we use throughout the text. Additional definitions will be given as they are needed.

A **simple graph** $G = (V, E)$ consists of a finite set of **nodes** (or vertices) V and a set of edges joining different pairs of distinct nodes. When the edges are defined by unordered pairs of nodes, the graph is **undirected**; otherwise it is a **directed graph**. All graphs discussed in this thesis are undirected simple graphs.

A graph is called **complete graph**, if there exist an edge joining each pair of distinct nodes. Such a graph is also called a **clique**. A complete undirected graph on of n vertices is denoted by K_n.

Two nodes i and j are **adjacent** if the edge, , and e_{ij} is said to be **incident** to the vertices i and j. The number of edges incident to a node is called the degree of the node. Notice the degree of every vertex in a complete graph K_n is $n-1$.

A node sequence $v_0, v_1, \ldots v_k$ with is a v_0 - v_k **walk** if for i = 1,...., k. Node v_0 is the origin of the walk and node v_k is the destination. Nodes

v_1,...., v_{k-1} are intermediate nodes. The walk has length k. The walk can also be represented by its edge sequence: e_1,..., e_k, where $e_i = (v_{i-1}, v_i)$. A walk is called a **path** if there are no node repetitions. A v_0 - v_k walk is **closed** if $v_k = v_0$. A closed walk is a **cycle** if and v_0, v_1,...., v_k is a path. A graph is **acyclic** if it contains no cycles. The **length** of a cycle is the number of edges in the cycle.

Two vertices u and v in V are **connected** in $G = (V, E)$ if there exists a $(u\text{-}v)$ path in G. Two vertices are in the same **component** of G if they are connected. $G = (V, E)$ is **connected** if it has exactly one component.

For, the graph $G' = (V', E')$ is a **subgraph** of G if and. G' is the subgraph **induced** by V' if. G' is a **spanning subgraph** if $V' = V$.

An acyclic graph is called a **forest**. A connected forest is a **tree**. A **spanning tree** of $G = (V, E)$ is a spanning subgraph that is a tree.

Graph Partitioning Problem

With the above definitions we here formalize the Clique Partition Problem or **Complete Graph Partitioning Problem** and its variations.

Given a graph $G = (V, E)$,

a clustering of G is a dividing

of V into , where and , . V_1, V_2,…, V_k are the components in the clustering, sometimes referred to as clusters or partitions. When G is a complete graph, we also call these components subcliques. We refer to the edge set E (V_1, V_2,... V_k) as the **partition set** or simply partition. We refer to the edge set $\delta(V_1, V_2,... V_k)$ as the

multi-cut set or simply multi-cut. When V is partitioned into just 2 subsets, V_1 and V_2, the resulting cut set $\delta(V_1, V_2)$ is simply called a **cut** of G. Notice here , .

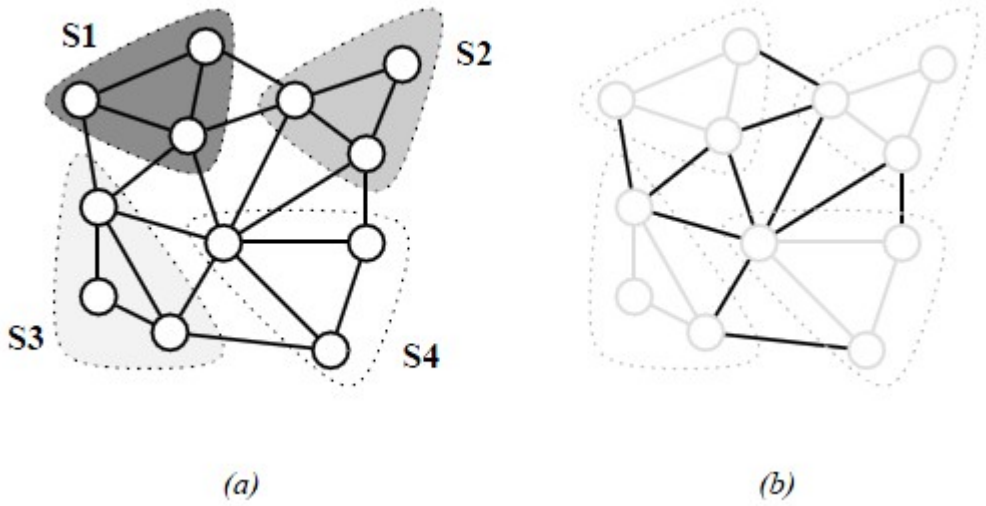

Fig1. (a) Balanced 4-way partition,
(b) Set of cut edges C cut from partition, $|C_{ij} = 11)$ [8]

3. CLASSES OF PARTITION PROBLEMS

Clique Partition Problem

Given a complete graph $G = (V, E) = K_n$, each edge is associated with an edge weight c_e. The Clique Partition Problem (CPP) is to partition the vertices V into sets of any number and any size, so as to minimize the total weight of the edges that have both endpoints in the same subclique.

2. Equipartition Problem

Given a graph $G = (V, E)$, each edge is associated with an edge weight c_e. The Equipartition problem is to partition the vertices into two sets of equal size, so as to minimize the total weight of the edges that have both endpoints in the same subclique.

3. K-way Equipartition Problem

Given a graph $G = (V, E)$, each edge is associated with an edge weight c_e. The k-way equipartition problem is to partition the vertices into k sets of equal size, so as to minimize the total weight of the edges that have both endpoints in the same subclique.

4. K-way Partition Problem

Given a graph $G = (V, E)$, each edge is associated with an edge weight c_e. The k-way partition problem is to partition the vertices into no more than k sets, so as to minimize the total weight of the edges that have both endpoints in the same subclique.

5. Capacitated Partitioning Problem

Given a complete graph $G = (V, E) = K_n$, each edge is associated with an edge weight c_e, each vertex is associated with a vertex weight d_v. The problem is to partition the vertices V into sets that satisfy certain weight restriction on vertex weights, so as to minimize the total weight of the edges that have endpoints in different subcliques.

Maxcut Problem

Given a graph $G = (V, E)$, each edge is associated with a non-negative edge weight c_e, we want to divide the vertices into 2 partitions so as to maximize the total weight of the edges that have endpoints in different subsets. Like many other graph optimization problems, all of the above partition problems are NP-hard, this

means that any exact algorithm known that can solve one of these problems needs CPU time that grows exponentially with the number of nodes in the graph unless *P* = *NP*. Thus one can only expect to solve such problems exactly for graphs with a limited number of nodes. The various techniques we are going to introduce, in particular, the integer programming approach, is trying to push up these limits. For a detailed introduction on complexity theory, including *P* and *NP*, we refer to Nemhauser and Wolsey [3]. Sipser [4] discusses this subject from a computer science point of view.

1. PREVIOUS ATTEMPTS

Various attempts were made to solve the classic clique partition problem. Here, we mention some unsuccessful attempts at heuristic solution to the partitioning problem.

6. Random Solutions

A simple tactic is to generate random solutions. A predefined value is kept as benchmark value. The solutions obtained randomly are compared with the predefined value. Values worse than the benchmark are rejected. In case a better value is obtained, the benchmark is updated. This process of generation is terminated after some predetermined time or a better value is reached.

This method is an n^2-procedure and is quite fast. However, this procedure is unsatisfactory for even moderately sized graphs, since there are generally few optimal or near-optimal solutions, which thus appear randomly with very low probabilities. Experience with 2-way partitions for a class of 0-1 matrix of size 32x32, for example, has indicated that there are typically 3 to 5

optimal partitions, out of a total partitions, giving a probability of success on any trial of less than 10^{-7}.

7. Max Flow-Min Cut

Another partition method is the Ford and Fulkerson max flow-min cut algorithm [5]. The graph is treated as a network in which edge costs correspond to maximum flow capacities between pairs of node. A cut is a separation of the nodes into two disjoint sets. The max flow-min cut algorithm states that the maximum flow values between any pair of nodes is equal to the minimal capacity of all the cuts which separate the two nodes. In our terminology, a cut is a 2-way partition, and the cut capacity is the cost of the partition. The Ford Fulkerson algorithm finds a cut with maximal flow, which is thus a minimal cost cut; this represents a minimum cost partition of the graph into two subsets of unspecified sizes.

There are several difficulties involved in using this algorithm for our partitioning problem. The most severe of these is the fact that the algorithm has no provision for constraining the sizes of the resultant subsets, and there seems to be no obvious way to extend it to include this. Thus if flow methods are used to perform a split, then further processing is necessary to make the resulting subsets the correct size. If the subsets are greatly different in size, then use of this algorithm will have produced essentially no benefit. Hence in spite of its theoretical elegance, the Ford and Fulkerson algorithm is not suitable for this application. (Note however, that since it does find the minimal cost unconstrained 2-way partition, the value it produces is a lower bound for solutions produced by any method.)

8. Clustering

A class of much more intuitive methods is based on identifying "natural clusters" in the given cost matrix—that is, groups of nodes which are strongly connected in some sense. For example, one can use very simple heuristics for building up clusters, based on collecting together elements corresponding to large values in the cost matrix. But again these methods do not in general include much provision for satisfying constraints on the sizes of the subsets, nor do they provide for systematic assignment of "stragglers" (nodes which do not obviously belong to any particular subset).

9. λ-0pting

Lin, working on the Travelling Salesman Problem [2], categorized a set of methods of improving given solutions by rearranging single links, double links, triplets, and in general, X links. He referred to a change involving the movement of X links as a X-change. If a configuration of the system is reached in which no X-change can be made which results in a decrease in cost, the configuration is said to be "X-opt." For the partitioning problem, an analogous operation is the interchange of groups of X points between a pair of sets. Thus a 1-change is the exchange of a single point in one set with a single point in another set. A configuration is then said to be "1-opt" if there exists no interchange of two points which decreases the cost of the partition. Experiments to evaluate 1-opting for 2-way partitions of 0-1 matrices (32 X 32) within which about one-half of the elements were nonzero, show that apparently optimal values can be achieved in about 10 percent of the trials while values within 1 or 2 of the optimal can be achieved in about 7.5 percent of cases.

It appears fruitless to extend X beyond 1 (1-opting is already an n2-procedure), or to extend 1-opting experiments to partitions into more than two

subsets, since more powerful methods have been developed. These methods are the topic of the next sections.

Two-way Uniform Partition

1. INTRODUCTION

The simplest partitioning problem which still contains all the significant features of larger problems is that of finding a minimal-cost partition of a given graph of 2n vertices (of equal size) into two subsets of *n* vertices each. The solution of the 2-way partitioning problem is the subject of this section. The solution provides the basis for solving more general partitioning problems.

Let *S* be a set of 2n points,
with an associated cost matrix *C* =
(c_{ij}), $i, j = 1,..2n$. We assume without loss of generality that *C* is a symmetric matrix, and that $c_{ii} = 0$ for all *i*. There is no assumption about non-negativity of the c_{ij}'s. We wish to partition *S* into two sets *A* and *B*, each with *n* points, such that the "external cost" is minimized.

In essence, the method is this: starting with any arbitrary partition *A, B* of *S,* try to decrease the initial external cost *T* by a series of interchanges of subsets of *A* and *B;* the subsets are chosen by an algorithm to be described. When no further improvement is possible, the resulting partition *A′, B′* is locally minimum with respect to the algorithm. We shall indicate that the resulting partition has a fairly high probability of being a globally minimum partition.

This process can then be repeated with the generation of another arbitrary starting partition *A, B,* and so on, to obtain as many locally minimum partitions as we desire.

Given *S* and (c_{ij}), suppose *A*, B** is a minimum cost 2-way partition. Let *A, B* be any arbitrary 2-way partition. Then clearly there are subsets with such that interchanging *X* and *Y* produces *A** and *B** as shown below.

The problem is to identify *X* and *Y* from *A* and *B,* without considering all possible choices. The process we describe finds *X* and *Y* approximately, by sequentially identifying their elements.

Let us define for each *at A,* an *external cost* E_a by

and an *internal cost* I_a by

Similarly, define E_b , I_b for each $b \in B$. Let $D_z = E_z - I_z$ for all $z \in S$; D_z is the difference between external and internal costs.

Lemma: Consider any $a \in A$, $b \in B$. *If a and b are interchanged, the gain (that is, the reduction in cost) is precisely* $D_a + D_b - 2c_{ab}$.

Proof: Let *z* be the total cost due to all connections between *A* and *B* that do not involve *a* or *b.* Then

$$T = z + E_a + E_b - c_{ab}$$

Exchange *a* and b; let *T'* be the new cost. We obtain

$$T = z + E_a + E_b + c_{ab}$$

and so

$$gain = \text{old cost} — \text{new cost} = T — T'$$
$$= D_a + D_b - 2c_{ab}$$

2. PHASE 1 PARTITION

In this subsection we present the algorithm for 2-way partitioning. First, compute the D values for all elements of S. Second, choose $a_i \in A$, $b_j \in B$ such that

$$g_i = D_{ai} + D_{bi} - 2c_{aibi}$$

is maximum; a_i and b_i correspond to the largest possible gain from a single interchange. (We will return shortly to a discussion of how to select a_i and b_i quickly.) Set a_i and b_i aside temporarily, and call them a_i' and b_i', respectively.

Third, recalculate the D values for the elements of $A - \{a_i\}$ and for $B - \{b_i\}$, by

$$D_x = D_x + 2c_{xai} - 2c_{xbi} \qquad x \in A - \{a_i\}$$
$$D_y = D_y + 2c_{ybi} - 2c_{yai} \qquad x \in A - \{b_i\}$$

The correctness of these expressions is easily verified: the edge (x, a_i) is counted as internal in D_x, and it is to be external in D_x', so c_{xai} must be added twice to make this correct. Similarly, c_{xbi} must be subtracted twice to convert (x, b_i) from external to internal.

Now repeat the second step, choosing a pair a_2', b_2' from $A - \{a_i\}$ and $B - \{b_i\}$ such that $g_2 = D_{a2} + D_{b2} - 2c_{a2'b2'}$ is maximum (a_1' and b_1' are *not* considered in this choice). Thus $g2$ is the additional gain when the points $a2$ and $b2$ are exchanged as well as a_1' and b_1'; this additional gain is maximum, given the previous choices. Set a_2' and b_2' aside also.

Continue until all nodes have been exhausted, identifying $(a_3', b_3'),\ldots,$ (a_n', b_n') and the corresponding maximum gains $g_3, \ldots, g_n$. As each (a', b') pair is identified, it is removed from contention for further choices so the size of the sets being considered decreases by 1 each time an

(a', &') is selected.

If $X = a_1', \quad a_2' \ldots, \quad a_k', \quad Y = b_1',$

$b_2', \ldots, b_k',$ then the decrease in cost

when the sets X and Y are interchanged is precisely $g_1 + g_2 + \ldots + g_k$. Of course . Note that some of the g_i's are negative, unless all are zero.

Choose k to maximize the

partial sum . Now if $G > 0$, a

reduction in cost of value G can be made by interchanging X and Y. After this is done, the resulting partition is treated as the initial partition, and the procedure is repeated from the first step.

If $G = 0$, we have arrived at a locally optimum partition, which we shall call a *phase 1 optimal partition.* We now have the choice of repeating with another starting partition or of trying to improve the phase 1 optimal partition. We shall discuss the latter option shortly. Following is a flowchart for the phase 1 optimization procedure.

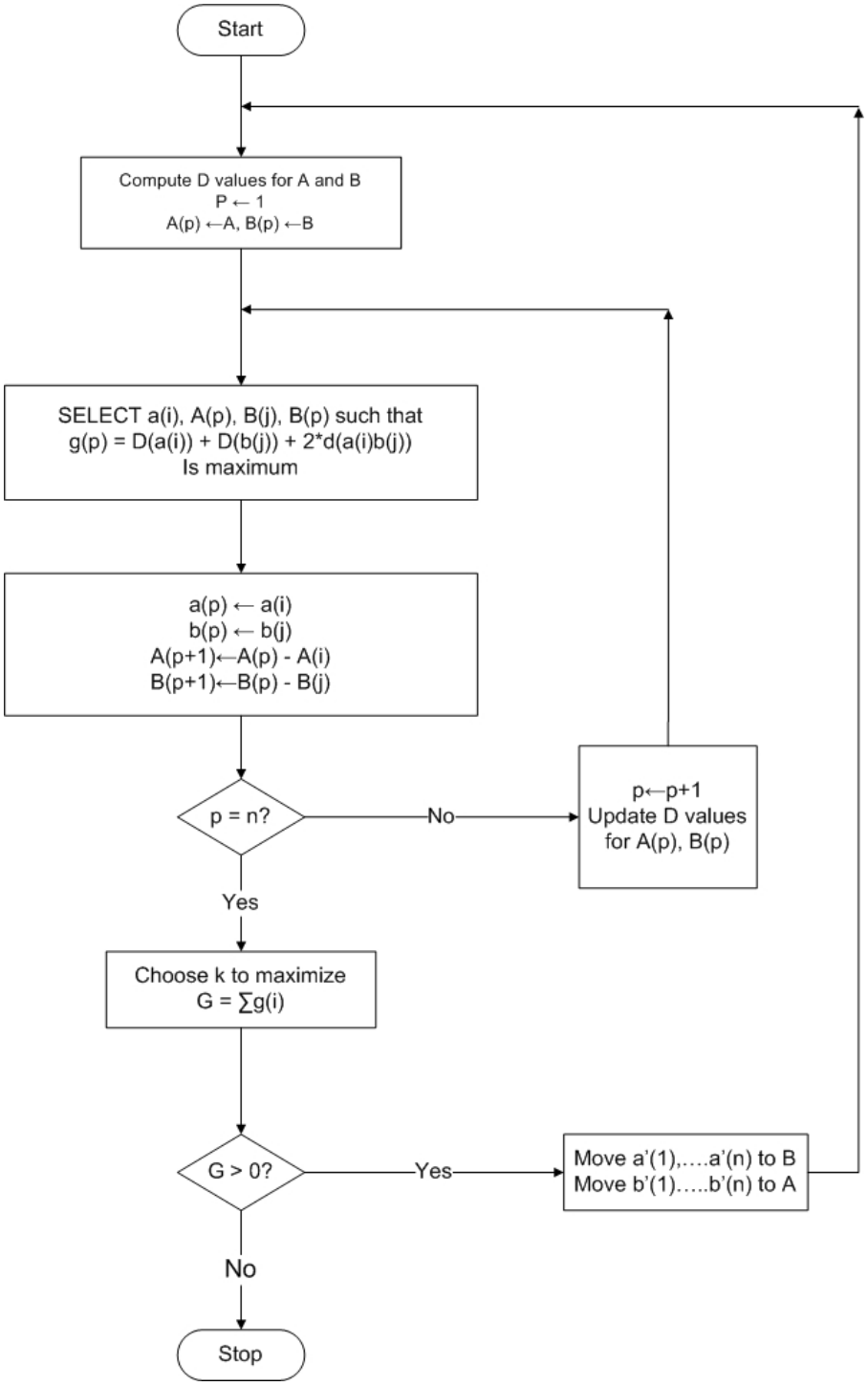

Fig2. Flowchart of phase 1 optimization procedure

3. EFFECTIVENESS OF THE PROCEDURE

One general approach to solving problems such as this one is to find the best exchange involving say λ pairs of points, for some λ specified in advance. The difficulty encountered is that use of a small value of λ is not sufficient to identify good exchanges, but the computational effort required grows rapidly as λ increases.

The procedure we have described sequentially finds an approximation to the best exchange of λ pairs. λ is not specified in advance, but rather is chosen to make the improvement as large as possible. This technique sacrifices a certain amount of power for a considerable gain in speed.

Since we construct a sequence of gains g_i, $i = 1,\ldots, n$, and find the maximum partial sum, the process does not terminate immediately when some g_i is negative. This means that the process can sequentially identify sets for which the exchange of only a few elements would actually increase the cost, while the interchange of the entire sets produces a net gain.

Numerous experiments have been performed to evaluate the procedure on different types of cost matrices. The matrices used have included *(i)* 0-1 matrices, with density of nonzero elements ranging from 5 percent to 50 percent, *(ii)* integer matrices with elements uniformly distributed on $[0, k]$, $k = 2,\ldots, 10$, *(iii)* matrices with clusters of known sizes and binding strength. Results on all of these matrices have been similar, so we shall only summarize them here. A more extended discussion may be found in [7].

A useful measure of the power of a heuristic procedure is the probability that it finds an optimal solution in a single trial. Suppose that p is the probability that a

phase 1 optimal solution found using a random starting partition is globally optimal. We have examined the behaviour of this probability as the size of the matrices involved is varied. Experiments show p is around 0.5 for matrices of size 30 X 30, 0.2 to 0.3 for 60 X 60, and 0.05 to 0.1 for 120 X 120. The functional behaviour of p is approximately $p(n) = 2^{-n/30}$.

These values are derived primarily from 0-1 matrices having about 50 percent l's (randomly placed). Experiments on matrices with lower densities of l's yield larger variances, but substantially identical mean values for p.

4. RUNNING TIME OF THE PROCEDURE

Let us define a *pass* to be the operations involved in making one cycle of identification of $(a_1', b_1'), \ldots, (a_n', b_n')$, and selection of sets X and Y to be exchanged. The total time for a pass can be estimated this way. First, the computation of the D values initially is an n^2-procedure, since for each element of S, all the other elements of S must be considered. The time required for updating the D values is proportional to the number of values to be updated, so the total updating time in one pass grows as

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$

which is proportional to n^2.

The dominant time factor is the selection of the next pair a_i', b_i' to be exchanged. The method we have used to perform this searching is to sort the D values so that

$$D_{a1} \geq D_{a2} \geq \ldots \geq D_{an}$$

and

$$D_{b1} \geq D_{b2} \geq .... \geq D_{bn}$$

When sorting is used, only a few likely contenders for a maximum gain need be considered. This is because when scanning down the set of D_a's and D_b's, if a pair D_{ai}, D_{bi} is found whose sum does not exceed the maximum improvement seen so far in this pass, then there cannot be another pair a_k, b_l with $k \geq i$, $l \geq j$, with a greater gain, (assuming $c_{ij} \geq 0$) and so the scanning can be terminated. Thus the next pair for interchange is found rapidly. Sorting is an n log n operation, so in this method, the total time required to sort D values in a pass will be approximately

$$n \log n + (n — 1) \log (n — 1) + ... + 2 \log 2$$

which grows as $n^2 \log n$.

To reduce the time for selection of an (a, *b)* pair, it is possible to use techniques which are faster than sorting, but which do not necessarily always give the maximum gain at each stage. For example, one method is to scan for the largest D_a and the largest D_b, and use the corresponding a and b as the next interchange. This method is essentially linear-time and would probably be implemented as part of the recomputation of the D values. It is best suited for sparse matrices, where the probability that $C_{ab} > 0$ is small. A slight extension, involving negligible extra cost, is to save the largest two or three $D_a's$ and $D_b's$, so that if the largest pair does not give the maximum gain (because c_{ab} is too large), then another can be tried. Experience indicates that three values are sufficient in virtually all cases, even for matrices with a relatively high percentage of nonzero entries. Use of this method reduces running time by about 30 percent in the present implementation, with very small degradation of power.

The number of passes required before a phase 1 optimal partition is achieved is small. On all matrix sizes tested at the time of writing (up to 360 points), it has

been almost always from 2 to 4 passes. On the basis of this experimental evidence, the number of passes is not strongly dependent on the value of n.

From the foregoing observations, it is possible to estimate the total running time of the procedure. If we use a method which sorts the D values at each stage (time proportional to $n^2 \log n$), then the running time should grow as $n^2 \log n$. If a fast-scan method is used, and the number of passes is constant, the running time should have an n growth rate; this is a lower bound.

For comparison, examination of all pairs of sets X and Y, and evaluation of the costs would require time proportional to

for large n. This function grows as $n^{3/2}4^n$.

Running times have been plotted in Fig. 2. The observed times have an apparent growth rate of about $n^{2.4}$, which is reasonably close to n^2. Although on the logarithmic plot this curve is close to linear over the range $n = 20$ to $n = 130$, it may actually be $n^2 \log n$; insufficient data is available to check this. All times are based on an implementation in FORTRAN G on an IBM System 360 Model 65.

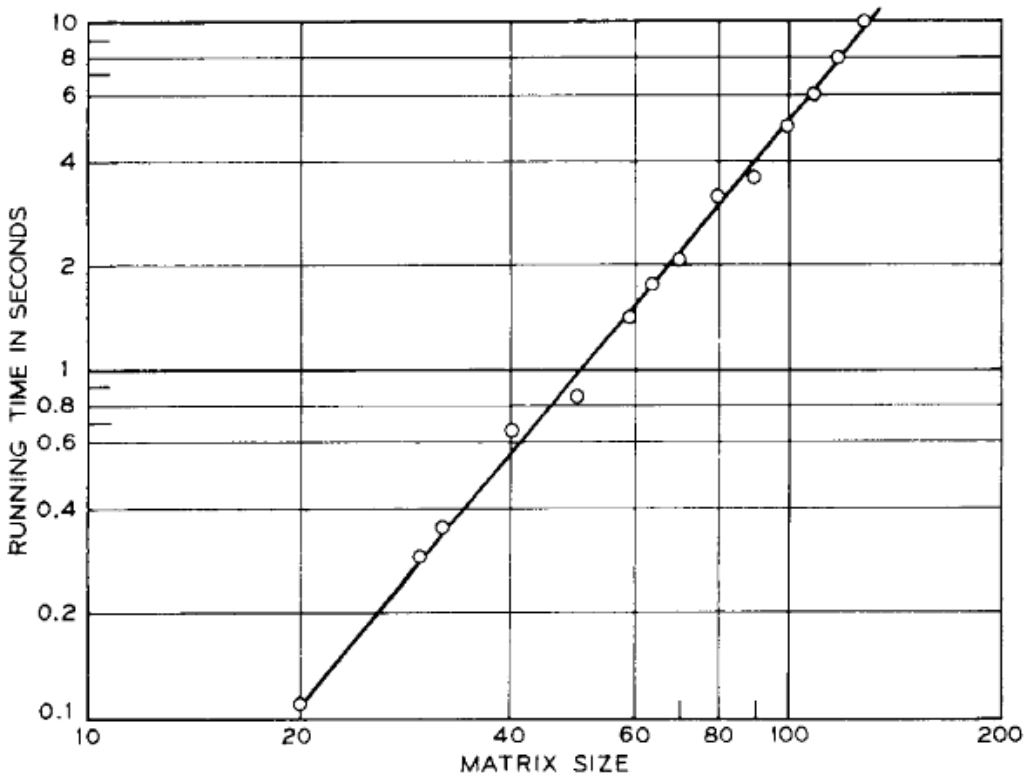

Fig3. Running Time of the Algorithm

5. IMPROVING PHASE 1 PARTITION

In this section, we discuss a method which might be used to improve the partition produced by the phase 1 procedure, which may not be globally optimal. The method suggested in this section is based heavily on experimental evidence, although there are quite plausible reasons for performing the particular set of operations. The basic idea is to perturb the locally optimal solution in what we hope is an enlightened manner, so that an iteration of the process on the perturbed solution will yield a further reduction in the total cost. If this tactic fails, nothing has been lost except some computation time, since the best solution seen so far is always saved.

Computer results for problems with up to 64 points suggest that whenever a phase 1 optimal solution is not globally optimal, $|X| = |Y| \approx n/2$. Roughly, this implies that if $|X|$ and $|Y|$ had been small compared to $n/2$, they would have been found by the process; it is only larger sets which are not identified all the time.

A successful heuristic to find the correct X and Y in this case is to find a phase 1 optimal partition for each of the sets A and B, say and (That is, find near-optimal partitions of A and of B separately). Recombine the 4 sets into 2, say and , and continue with phase 1 optimization. If our expectation is correct, the new X and Y will be small, and thus readily identified by the phase 1 process.

When A is split into A_1, $A2$ and B into B_1, B_2 there are two ways in which the smaller sets can be recombined. A series of tests was made on matrices of moderate size (up to 64 X 64), in which both possible recombination were done, generating three phase 1 optimal values for each starting partition. For matrices of size 32 X 32, the apparent optimal value was observed at least once in each triple of values, for a large number of cases. With matrices of size 64 X 64, there were occasional failures.

It might be noted that the extra time involved for the recombination approach is three times that required to do a completely new partition from a random start, assuming an n^2-procedure.

It is possible to estimate whether a particular improvement tactic is profitable or not in the following way. Suppose that some method increases the probability of finding an optimal partition from p to p', while it increases the running time from t to t'. Then in a fixed amount of time, it is possible to do k trials of the basic procedure, and kt/t' trials of the improved method. The corresponding probabilities of achieving an optimal solution are $1 - (1 - p)^k$ and $1 - (1 - p')^{kt/t'}$

respectively. The improved method is then desirable if the second expression is greater than the first; by simple manipulation, this condition becomes

$$(1 - p') < (1 - p)^{t'/t}$$

On the basis of the numerical values in this section, it may be useful to try the recombination method.

6. PARTITIONING UNEQUAL SIZED SETS

It is simple to modify the procedure to partition a set *S* with *n* elements into two sets of specified sizes n_1 and n_2 ($n_1 + n2 = n$). Assume $n_1 < n_2$. Then restrict the maximum number of pairs that can be exchanged in one pass of the procedure to n_1. All other operations are performed on all elements of each set. (The starting partition is into two sets, of ri, and n2 elements respectively.)

Suppose we wish to partition *S* into two sets, such that there are at least n_1 elements and at most n_2 elements in each subset; $n_1 + n_2 = n$, but they are not specified further.

The procedure is easily modified to handle this sort of constraint by the addition of "dummy" elements. These are elements which have no connections whatsoever; that is, they have zero entries in the cost matrix wherever they appear. Add $2n_2 - n$ dummies so *S* has $2n_2$ elements, and perform the procedure on it. The resulting partition will assign the dummy elements to the two subsets so as to minimize the external cost; at this point the dummies are discarded, leaving a partition into two subsets that satisfy the size constraints given.

7. MODIFIED TASK PERFORMED

The task performed was to partition a given network graph using the above algorithm considering the distance of separation of the nodes. The network chosen was a real network called ARPANET which is also regarded as the origin of the Internet. The number of node computers in it is over 40. However many computers were geographically very close to each other. For example, many universities like Harvard, MIT and Cambridge, all had their own computers that acted as separate nodes in ARPANET. However, since these computers are in very high proximity with each other, the effective distance between them is zero. In such cases, these nodes are considered as one. The resultant number of nodes turned out to be 28. The primary objective was to partition these 28 nodes into 2 sets, each having 14 nodes. Also, the total external distance between the two sets was maximal.

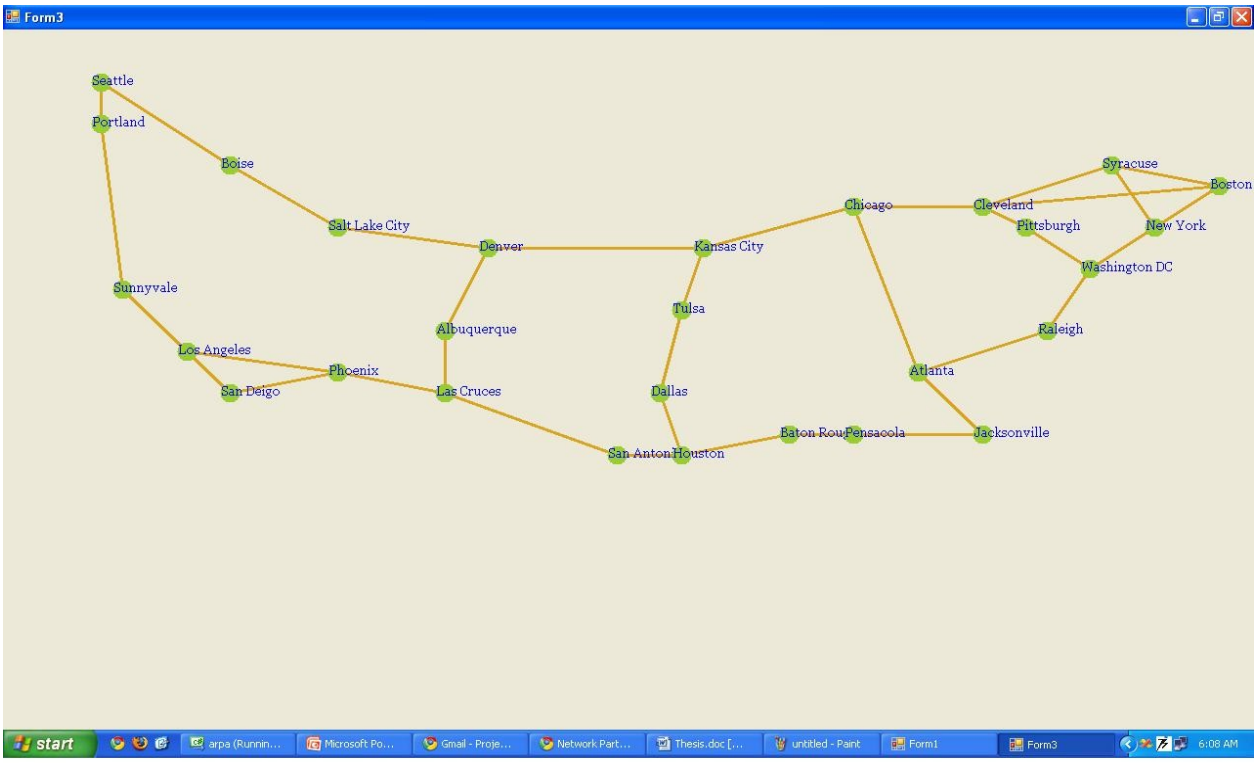

Fig4. ARPANET (nodes named after places)

Although the algorithm adopted was fundamentally the same as the classic Kernighan Lin's Two-way Partition Algorithm, certain modifications had to be done in order to account for the distance instead of the cost of the edges. The main difference between the two cases was consideration of distance while partitioning instead of the cost of the edges. In the classic problem, the cost of partition was supposed to be minimized. However in our case, the partition should have been created in such a way that the nodes belonging to different partition should be as far as possible. In other words, the distance between any two nodes belonging to different partitions should be as high as possible. That is, all the paths cost associated with the external edges (in this case the distance) should be maximized. This is where the original had to be modified to address the current problem.

Yet another difference between the two cases is that the classical cost based graph partition problem is presumes the graph to be completely connected. Whereas in our distance problem, the graph need not be completely connected. As a result, the shortest distance of each node to other nodes is calculated to have an essence of complete connectivity.

In Cost Problem, partitioning into unequal-sized sets is much easier. It can easily be accomplished by adding "dummy" elements or nodes to the existing graph. Obviously the weight and edge cost for the dummy nodes would be zero. The number of dummy elements should make up for the disproportion, i.e. the no. of dummy nodes should be equal to (n_2-n_1). In Distance Problem Partitioning into unequal-sized sets is not as simple because dummy nodes with no edges or edges with distance zero as same as counting one cities number of times.

Modified Lemma: Consider any $a \in A$, $b \in B$. If a and b are interchanged, the gain (that is, the reduction in cost) is precisely $Da + D_b + 2d_{ab}$.

Proof: Let z be the total cost due to all connections between A and B that does not involve either a or b. Then

$$T = z + E_a + E_b - d_{ab}$$

Exchange a and b ; Let T' be the new cost. We obtain

$T' = z + I_a + I_b + d_{ab}$

and so

gain = new cost - old cost = $T' — T$

$= D_a + D_b + 2d_{ab}$

Cost Problem cannot be converted to Distance Problem directly because

- The Total External Cost of the Cost Problem is minimized whereas in Distance Problem the Total External Distance is maximized
- Due the above difference, all the formulae in the procedure are required signs changes

I. Distance: $D_z = I_z — E_z$

Cost: $D_z = E_z — I_z$

Difference is changed to make D_z with positive sign in Gain

II. Distance: Gain = $D_a + D_b + 2d_{ab}$

Cost: Gain = $D_a + D_b – 2c_{ab}$

Gain value is changed to because here Gain will be decreasing I_z and increasing E_z and d_{ab} is added twice to make to compensate node transfer's distance.

III. Distance: $D_x' = D_x - 2d_{xai} + 2d_{xbj}$; $x \in A - \{a_i\}$

Cost: $D_x' = D_x + 2c_{xai} - 2c_{xbj}$; $x \in A - \{a_i\}$

Updating of D_z in Distance Problem is as above because the $D_z = I_z - E_z$

- Also the Cost Problem is for completely connected graph but Distance Problem does not contain completely connected Network. So, the shortest distance of each node to other nodes is calculated to have an essence of complete connectivity.
- In Cost Problem Partitioning into Unequal-Sized Sets is easier
  - By taking (n_2-n_1) dummy elements with cost zero

In Distance Problem Partitioning into Unequal-Sized Sets is not so simple

  - Dummy nodes with no edges or edges with distance zero as same as counting one cities number of times

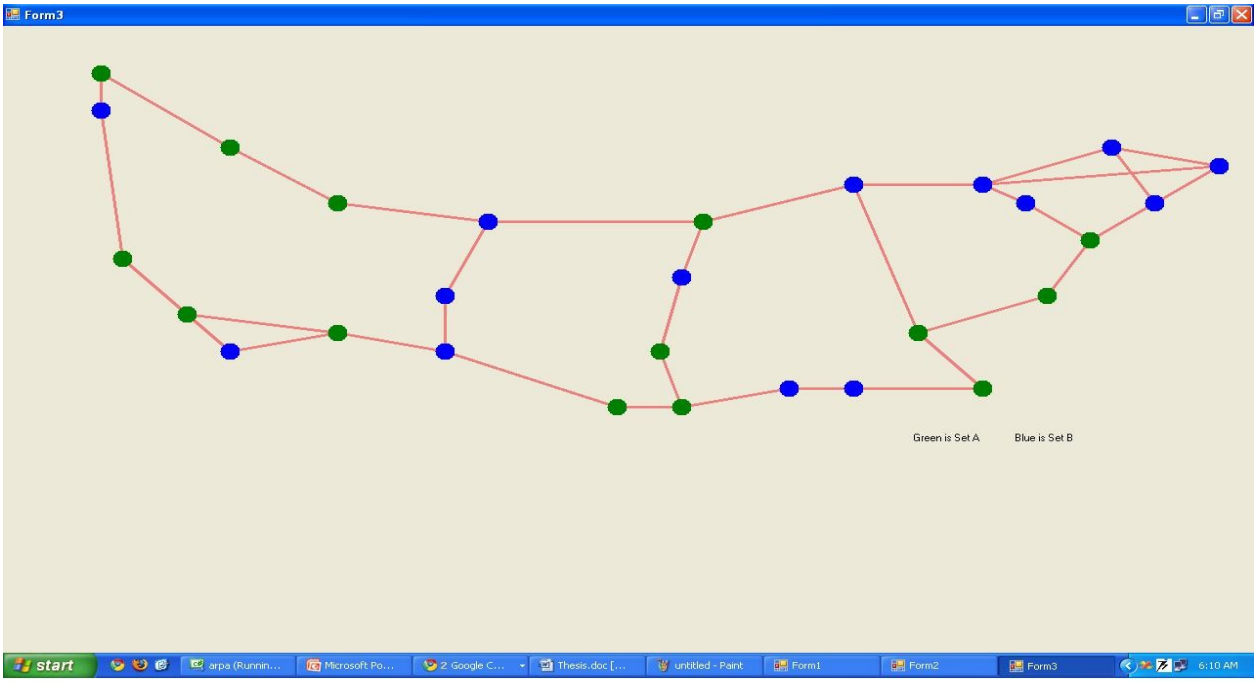

Fig5. Initial Random 2-way Partition of the ARPANET

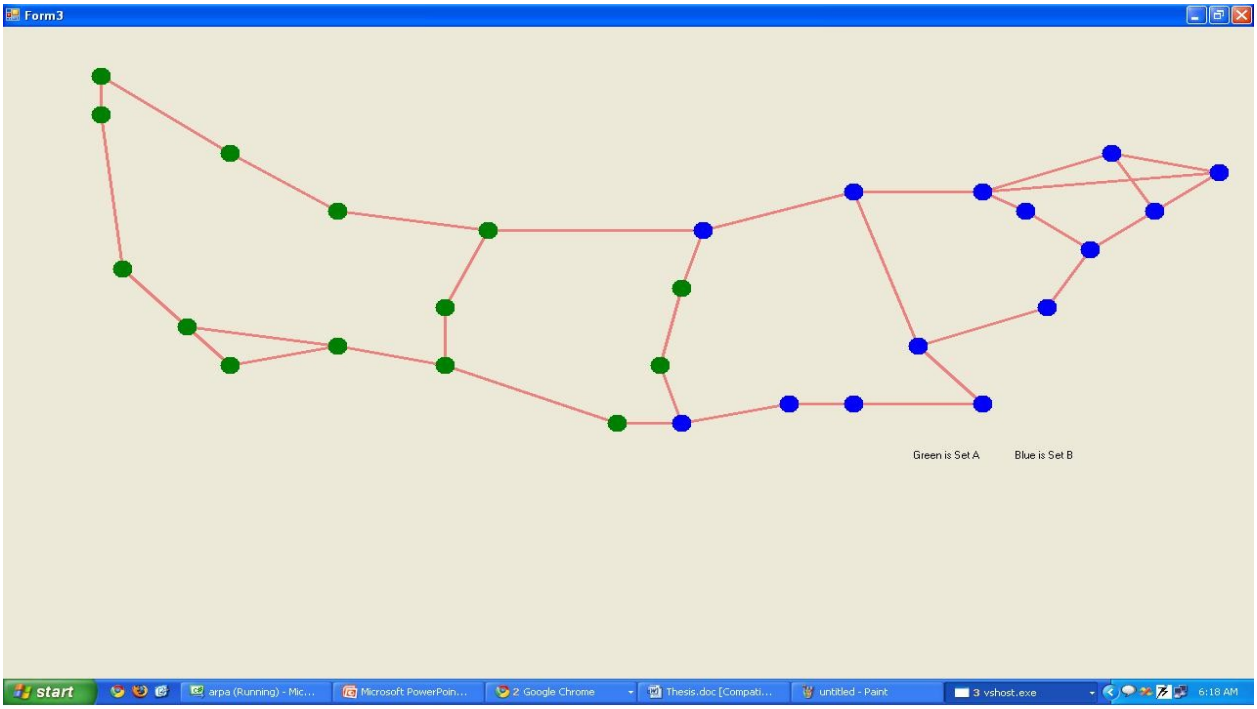

Fig6. Final 2-way partition

Biggest Problem arose during implementation was that after partitioning, the graph contained two three unconnected parts or few nodes were isolated from the remaining nodes of partition.

This Problem was solved by checking at the end of every Phase-1 Procedure that the graph doesn't contain unconnected parts and if it does then Phase 1 Procedure is applied again or new random partition is generated again and then Phase 1 Procedure is applied.

The algorithm is recursively used to obtain a 4-way partition. That is, the graph is partitioned into 4 sets, each containing 28/4 = 7 nodes. The resultant graphs are as follows:

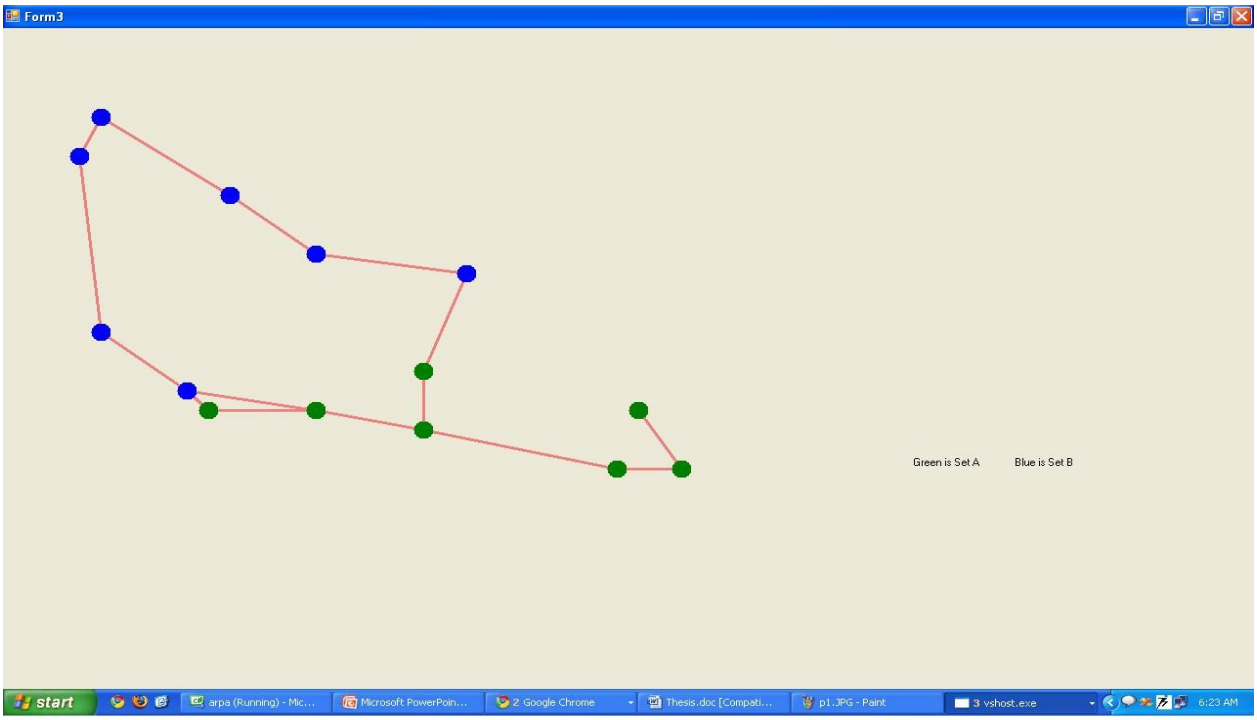

Fig7. Part 1 of 4-way partition

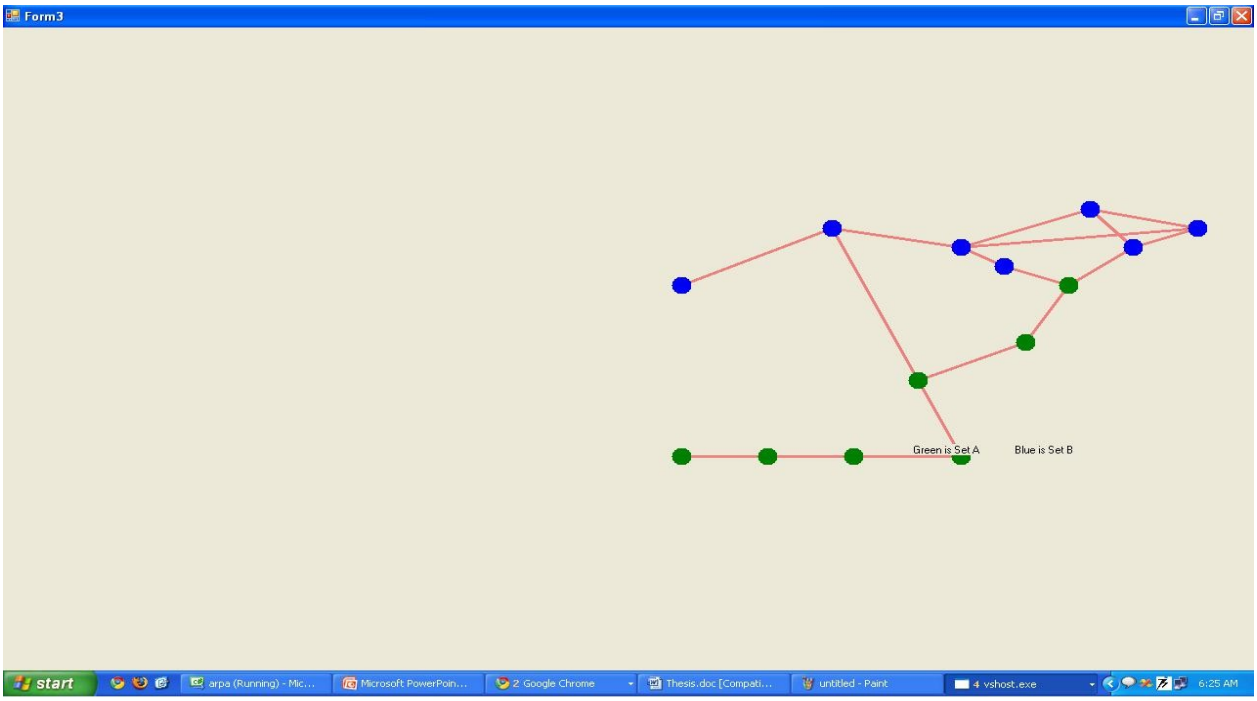

Fig8. Part 2 of 4-way Partition

CONCLUSION

In this paper, we implemented the classic 2-way graph partition algorithm in order to partition a real network called ARPANET. The problem is basically a NP-Hard problem and cannot be solved by direct or exhaustive search algorithms. Instead, heuristic algorithmic approach was adopted to solve the problem in feasible time. The method considered was primarily stated by Kernighan and Lin in [1]. However, the prime difference was that in our case, the edges had distances associated with them instead of cost. Had it been cost problem, the total external cost was supposed to be minimized. Since it was a distance problem, the algorithm involved maximization of external distance which meant that node computers those are in different partitions were as far from each other as possible. Various problems were faced and the original work needed to be modified. The foremost problem was to modify the base equation in order to accommodate the distance instead of cost. Also, the classical problem considered completely connected graph. However, experience shows in real life, network graphs cannot always be completely connected. Hence, the shortest distance of each node to other nodes is calculated to have an essence of complete connectivity. The graph could not be further divided because the remaining number of nodes is odd and hence the algorithm would fail to create 2 equipartitions. Also, dummy nodes cannot be added in distance problem as a node with zero weight and zero distance from another node would effectively mean the same node over again. The essence is that the equipartition algorithm can be used while the current set has even number of nodes. The running of the algorithm was much better than the counterparts. The

effective complexity turned out to be $n^{2.4}$ which is quite fast given the number of nodes. Finally, the algorithm was recursively used to further partition the 2 subsets of ARPANET to give way to 4 final partitions.

References

1. B. W. Kernighan and S. Lin, *"An efficient Heuristic Procedure for Partitioning Graphs"*, 1969
2. Larry Roberts, ARPANET
3. George L. Nemhauser and Laurence A. Wolsey, *"Integer and Combinatorial Optimization"*. John Wiley & Sons, 1999
4. Michael Sipser. *"Introduction to Theory of Computation"*, 1996
5. L.R. Ford and D.R. Fulkerson, *"Flow in Networks, Princeton"*, New Jersey, 1962
6. S. Lin, *"Computer Solutions to Travelling Salesman Problem"*, 1965
7. B.W. Kernighan, "*Some Graph Partitioning Problems Related to Program Segmentation*", 1969
8. B. Chamberlain, "*Graph Partitioning Algorithms for Distributed Workloads of Parallel Computations*", 1998