DESIGN AND IMPLEMENTATION OF DISEASE RECOGNITION SYSTEM USING MINIMUM SPANNING TREES

By

Reema Mukhtar
2006-NUST-MS PhD-CSE(E)-02
MS(CSE)

Advisor
Brig. Dr. Muhammad Younus Javed

Thesis submitted in partial fulfillment of the requirements for the degree

of

Masters of Science
In
Computer Software Engineering
College of E&ME
National University of Science and Technology, Rwp
2009

ACKNOWLEGMENTS

I am grateful to Allah Almighty for always bestowing his mercy upon me and guiding me and enabling me to complete this thesis.

First of all I would like to thank my supervisor Brig. Dr. Muhammad Younus Javed for his invaluable guidance and advices throughout this thesis. His supervision and support helped a lot in the completion of this thesis. Also, I am thankful to all those individuals who helped me at different stages of this thesis.

I would also like to thank the UCI Machine Learning Repository, for the disease datasets, which proved very helpful in testing the system.

And last but not least, I thank my family and friends for all their support.

Reema Mukhtar.

Dedicated to my parents, sisters and Junaid;
and to Mahnoor, Amal, Eshal and Raneem

Abstract

Accurate and reliable Pattern Recognition is critical in many fields such as medicine, biometrics, character recognition, speech recognition, bioinformatics etc. Pattern recognition attempts to instill in computers some of the cognitive abilities of humans. Based on some already known data samples, the system is trained and it is then able to recognize and classify objects by using the information it has learnt. The samples are represented by different features. In real world, these features may be hundreds and thousands in number. Some of these features may be redundant which may not provide any help in classifying the objects. When these data samples are gathered, noise may be added in those feature values which can actually play a negative role by classifying incorrectly. Also, the process of training and recognition will take a lot of time when all these features are used. Thus feature subset selection i.e. the process of selecting a smaller subset of features from the entire feature set, so that maximum accuracy can be achieved in classification in a reasonable amount of time, is an important area of research.

This thesis describes feature subset selection and its implementation by using Minimum Spanning Trees. Graphs are built on the sample training data with the nodes of the graph equal to the number of data points. The edges of the graph are constructed by calculating the euclidean distance between samples using some features. Minimum spanning trees are then built on the graph for different feature subsets. These trees are then evaluated through a criterion function to determine the best spanning tree which will result in the best accuracy. The criterion function chooses such spanning trees which have dense clusters of samples of one class distinctively separated from clusters of other classes. This ensures good accuracy for recognition through the nearest neighbor method. For determining the recognition and classification performance of the system, three data sets from the field of medicine are used. Maximum classification accuracy of 96% is achieved using these data sets.

The main phases of the implementation are training, feature subset selection, recognition and classification. For feature subset selection, minimum spanning trees are used to select the best feature subset that provides good accuracy. For recognition and classification, k-nearest neighbor approach is used in which the user can specify the desired value for 'k'. The True Positive and False Positive rates are then calculated to assess the accuracy of the system.

Table of Contents

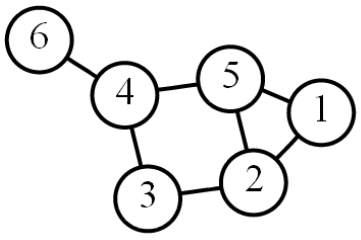
List of Figures

Chapter 1 : Introduction

1.1 Graphs

Graphs are mathematical structures used to model pair-wise relations between objects from a certain collection. A graph therefore, refers to a collection of vertices or 'nodes' and a collection of *edges* that connect pairs of vertices.

A graph G is represented as G (V, E) where V is the set of vertices or nodes in the graph and E is the set of edges connecting these nodes.



1-1: An example of a graph

1.1.1 Data Structures for representing Graphs

There are different ways to store graphs in a computer system. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. List structures are often preferred for sparse graphs as these have smaller memory requirements. Matrix structures, on the other hand, provide faster access for some applications but can consume huge amounts of memory since these require N^2 memory for a graph on N nodes.

1.1.1.1 List Structures

- **Incidence List**

The edges are represented by an array containing pairs of vertices that make the edge and possibly weight and other data. These pairs of vertices are ordered in case of a directed graph. Vertices connected by an edge are said to be *adjacent*.

- **Adjacency List**

It is similar to the incidence list in which each vertex has a connected list of vertices it connects to. This is not very efficient since it causes redundancy in an undirected graph: for example, if vertices A and B are adjacent, A's adjacency list contains B, while B's list contains A. Adjacency queries are faster, at the cost of extra storage space.

1.1.1.2 Matrix Structures

- **Incidence Matrix**

The graph is represented by a matrix of size $|V|$ (number of vertices) by $|E|$ (number of edges) where the entry [vertex, edge] contains the edge's endpoint data. This entry value may be 1 for connected and 0 for not connected vertices.

- **Adjacency matrix**

This is an n by n matrix A, where n is the number of vertices in the graph. If there is an edge from some vertex x to some vertex y, then the element $a_{x,y}$ is 1, otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

- **Laplacian matrix or Kirchhoff matrix or Admittance matrix**

This is defined as $D - A$, where D is the diagonal degree matrix and A is the adjacency matrix. It explicitly contains both adjacency information and degree information.

- **Distance matrix**

This is a symmetric n by n matrix D whose element $d_{x,y}$ is the length of the shortest path between x and y; if there is no such path, then $d_{x,y}$ = infinity. It can be derived from powers of A: $d_{x,y} = \min\{n \mid A^n[x,y] \neq 0\}$.[15]

1.1.2 Problems in Graph Theory

1.1.2.1 Subgraphs

A common problem, called the *subgraph isomorphism problem*, is to find a fixed smaller graph in a given graph. One reason to be interested in such a question is that many graph properties are hereditary for subgraphs, which means that a graph has the property if and only if all subgraphs, or all induced subgraphs, have it too. Unfortunately, finding maximal subgraphs of a certain kind is often an NP-complete problem. Finding the largest complete graph is called the clique problem which is also NP-complete.

1.1.2.2 Induced Sub-graphs

A similar problem is to find induced subgraphs in a given graph. Some important graph properties are hereditary with respect to induced subgraphs, which means that a graph has a property if and only if all induced subgraphs also have it. Finding maximal induced subgraphs of a certain kind is also often NP-complete [14]. Finding the largest edgeless induced subgraph, or independent set, called the independent set problem, is also an NP-complete problem.

1.1.2.3 Minor (Sub-contraction)

The *minor containment problem* is to find a fixed graph as a minor of a given graph. A minor or **subcontraction** of a graph is any graph obtained by taking a subgraph and contracting some (or no) edges [16]. Many graph properties are hereditary for minors which means that a graph has a property if and only if all minors have it too.

1.1.2.4 Graph coloring

Graph Coloring is the problem of choosing minimum number of colors to color the vertices of the graph such that no two adjacent vertices have the same color. Different variations of this problem have been defined, for example:

- The four-color theorem
- The strong perfect graph theorem
- The Erdős–Faber–Lovász conjecture (unsolved)
- The total coloring conjecture (unsolved)

- The list coloring conjecture (unsolved)

1.1.2.5 Route problems

- ***Hamiltonian path and cycle problems***

In the mathematical field of graph theory the *Hamiltonian path problem* and the *Hamiltonian cycle problem* are problems of determining whether a Hamiltonian path or a Hamiltonian cycle exists in a given graph (whether directed or undirected) [14]. Both problems are NP-complete. The problem of finding a Hamiltonian cycle or path is in FNP.

The Hamiltonian path problem for graph **G** is equivalent to the Hamiltonian cycle problem in a graph **H** obtained from **G** by adding a new vertex and connecting it to all vertices of **G**.

The Hamiltonian cycle problem is a special case of the traveling salesman problem, obtained by setting the distance between two cities to a finite constant if these are adjacent and infinity otherwise.

A randomized algorithm for Hamiltonian path, that is fast on most graphs, is described as: start from a random vertex, and continue if there is a neighbor not visited. If there are no more unvisited neighbors, and the path formed isn't Hamiltonian, pick a neighbor uniformly at random, and rotate using that neighbor as a pivot. Then, continue the algorithm at the new end of the path [14].

- ***Minimum spanning tree***

This is the problem of finding a tree of a graph G such that it spans all the vertices (nodes) of the graph with the minimum cost edges. Minimum spanning trees will be described in detail in later sections.

- ***Route inspection problem (also called the "Chinese Postman Problem")***

The *Chinese Postman Problem, Postman Tour* or *Route Inspection problem* is to find the shortest closed trail (circuit) that visits every edge of a (connected) undirected graph. When the graph has an Eulerian circuit, that circuit is an optimal solution [16].

- ***Bottleneck Traveling Salesman***

The *Bottleneck Traveling Salesman Problem* (bottleneck TSP) is a problem in discrete or combinatorial optimization. It is the problem of finding the Hamiltonian cycle in a weighted graph with the minimum weight of the most costly edge of the cycle. The problem is known to be NP-hard [16].

There are certain variations of Bottleneck Traveling Salesman problem. In an *Asymmetric Bottleneck TSP*, there are cases where the weight from node *A* to *B* is different from the weight from B to A (e. g. travel time between two cities with a traffic jam in one direction). The *Euclidean Bottleneck TSP*, or planar bottleneck TSP, is the bottleneck TSP with the distance being the ordinary Euclidean distance. The problem still remains NP-hard, however many heuristics work better.

If the graph is a metric space then there is an efficient approximation algorithm that finds a Hamiltonian cycle with maximum edge weight being no more than twice the optimum.

- ***Euclidean Traveling Salesman***

The Euclidean Traveling Salesman (TSP) is a problem in combinatorial optimization. Given a list of cities and their pair-wise euclidean distances, the task is to find a shortest possible tour that visits every city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as genome sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents traveling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSP problems. Especially, it is plausible that the worst case running time for an algorithm for TSP depends exponentially on the number of cities, so that even some instances with only dozens of cities cannot be solved exactly [16].

- ***Shortest Path Problem***

The *Shortest Path Problem* is the problem of finding a path between two vertices (or nodes) such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel through that segment.

Formally, given a weighted graph (that is, a set V of vertices, a set E of edges, and a real-valued weight function $f : E \to \mathrm{R}$), and one element v of V, find a path P from v to each v' of V so that

$$\sum_{p \in P} f(p)$$

is minimal among all paths connecting v to v' [16].

The problem is also sometimes called the *single-pair shortest path problem*, to distinguish it from the following generalizations:

- The *single-source shortest path problem*, in which we have to find the shortest paths from a source vertex v to all other vertices in the graph.
- The *single-destination shortest path problem*, in which we have to find the shortest paths from all vertices in the graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the edges in the graph.
- The *all-pairs shortest path problem*, in which we have to find the shortest paths between every pair of vertices (v, v') in the graph.

These generalizations require significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.

- ***Euler Circuit***

The problem of finding an *Euler Circuit* in a graph G is to find a path in which every vertex has an even degree and G contains an Euler trail connecting two vertices x and y where x and y are the only two vertices of odd degree.

- ***Longest Path***

The *Longest Path* between two vertices in a connected graph *G* is a path of maximum length between the vertices.

- ***Prize Collecting Traveling Salesman***

The PCTS problem is described as: A salesman travels between pairs of cities at a cost depending only on the pair and gets a prize in every city that he visits and pays a penalty to every city that he fails to visit. The problem is to minimize his travel costs and net penalties, while visiting enough cities to collect a prescribed amount of prize money.

- ***Rural Postman***

The Rural Postman Problem (RPP) is a generalization of the Chinese Postman Problem in which a subset of the edges E_r (called required edges) has to be traversed. It is the problem of determining a minimum–weight closed route traversing each edge in E_r, at least once.

- ***Shortest Weight-Constrained Path***

Given a directed graph whose arcs have an associated cost, and associated weight, the weight constrained shortest path problem (WCSPP) consists of finding the least-cost path between two specified nodes, such that the total weight along the path is less than a specified value.

- ***Stacker-Crane***

The *Stacker-Crane Problem (SCP)* is a sequencing problem, arising in scheduling and transportation that consists of finding the minimum cost cycle on a mixed graph with oriented arcs and un-oriented edges. Feasible solutions must traverse all the arcs.

Approximation algorithms are known to provide a fixed worst-case bound if the triangle inequality holds

- ***Time Constrained Traveling Salesman Problem***

The time constrained traveling salesman problem is a variation of the familiar traveling salesman problem that includes time window constraints on the time a particular city, or cities, may be visited.

- ***Vehicle Routing Problem***

The *Vehicle Routing Problem (VRP)* is a combinatorial optimization and nonlinear programming problem seeking to service a number of customers with a fleet of vehicles. VRP is an important problem in the fields of transportation, distribution and logistics. Often the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. Implicit is the goal of minimizing the cost of distributing the goods. Many methods have been developed for searching for good solutions to the problem, but for all but the smallest problems, finding global minimum for the cost function is computationally complex.

1.1.2.6 Eulerian Paths and Circuits

To find the Eulerian Path and circuit in a graph, following statements must be considered:

- Suppose we have a connected graph G = (V, E), All vertices in G have even degree
- G consists of the edges from a disjoint union of some cycles, and the vertices from these cycles.
- G then has an Eulerian circuit.

A semi-Eulerian path (a path which is not closed but uses all edges of G just once) exists if and only if G is connected and exactly two vertices have non-even valence.

If a graph is Eulerian, then a Eulerian path visits every edge, and so the solution is to choose any Eulerian path.

If the graph is not Eulerian, it must contain vertices of odd degree. As already proved in graph theory, there must be an even number of these types of vertices. We make the graph Eulerian by doubling the paths that connect these vertices in pairs. We choose the pairs such that the total distance covered by all paths that connect these vertices are as small as possible. Now the solution is an Eulerian path for this new graph.

Some other path problems related to graphs are:

- Shortest path problem
- Steiner tree
- Three-cottage problem
- Traveling salesman problem (NP-complete)

1.1.2.7 Network Flow

These are the problems of finding a flow or path in the graph, such that the cost of that path is maximized or minimized. There are numerous problems arising especially from applications that have to do with various notions of flows in networks.

1.1.2.8 Visibility Graph Problems

A *Visibility Graph* is a graph of inter-visible locations. Each node or vertex in the graph represents a point location, and each edge represents a visible connection between these (that is, if two locations can see each other, an edge is drawn between these) [16].

Special classes are visibility graphs for points in the plane, in particular, within a polygon. The polygon may or may not have *holes* (obstructions within the plan). A major open problem in this area is to characterize the visibility graphs of polygons.

In addition to theoretical problems, visibility graphs also have practical uses, for example, to calculate the placement of radio antennas, or as a tool used within architecture and urban planning through visibility graph analysis. Visibility graphs are also used in mobile robotics as a motion planning tool when the geometry of the environment is known, although robots have been designed to collect information as these explore the environment using ultrasound sensors, which can then be turned into a visibility graph of recognizable known locations.

1.1.2.9 Covering Problems

Covering problems are specific instances of subgraph-finding problems, and these tend to be closely related to the clique problem or the independent set problem. Variations include:

- Set cover problem
- Vertex cover problem

1.1.3 Applications of Graphs

Applications of graph theory are primarily, but not exclusively, concerned with labeled graphs and various specializations of these.

Structures that can be represented as graphs are ubiquitous, and many problems of practical interest can be represented by graphs. The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page *A* to page *B* exists if and only if *A* contains a link to *B*. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. These are either directly used or properties of the rewrite systems (e.g. confluence) are studied.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pair-wise connections have some numerical values [15]. For example if a graph represents a road network, the weights could represent the length of each road.

Graphs have widely been used in the area of artificial intelligence, to model any problem in which relations exist between different elements of interest.

Networks have many uses in the practical side of graph theory, network analysis (for example, to model and analyze traffic networks). Within network analysis, the definition of the term "network" varies, and may often refer to a simple graph.

Many applications of graph theory exist in the form of network analysis. These can be split, broadly, into three categories. First: an analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist, and the production of useful ones for various domains remains an active area of research. Second: an analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it. Third: an analysis of dynamical properties of networks.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms, for example, Franzblau's shortest-path (SP) rings [15]. In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges represent bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software [15].

1.2 Trees

In graph theory, a **tree** is a graph in which any two vertices are connected by *exactly one* path. Alternatively, any connected graph with no cycles is a tree. A **forest** is a disjoint union of trees. Trees are widely used in computer science data structures such as binary search trees, heaps, tries, Huffman trees for data compression, etc.

A tree is an undirected simple graph *G* that satisfies any of the following equivalent conditions:

- *G* is connected and has no cycles.
- *G* has no cycles, and a simple cycle is formed if any edge is added to *G*.
- *G* is connected, and it is not connected anymore if any edge is removed from *G*.

- G is connected and the 3-vertex complete graph K_3 is not a minor of G.
- Any two vertices in G can be connected by a unique simple path.

If G has finitely many vertices, say n of these, then the above statements are also equivalent to any of the following conditions:

- G is connected and has $n - 1$ edges.
- G has no simple cycles and has $n - 1$ edges.

An undirected simple graph G is called a *forest* if it has no simple cycles.

A **directed tree** is a directed graph which would be a tree if the directions on the edges were ignored.

A tree is called a **rooted tree** if one vertex has been designated as the *root*, in which case the edges have a natural orientation, *towards* or *away* from the root. The *tree-order* is the partial ordering on the vertices of a tree with $u \leq v$ if and only if the unique path from the root to v passes through u. A tree which is a subgraph of some graph G is a **normal tree** if the ends of every edge in G are comparable in this tree-order [17]. Rooted trees, often with additional structure such as ordering of the neighbors at each vertex, are a key data structure in computer science. In a context where trees are supposed to have a root, a tree without any designated root is called a **free tree**.

A **polytree** has, at the most, one undirected path between any two vertices. In other words, a polytree is a directed acyclic graph (DAG) for which there are no undirected cycles either.

A **labeled tree** is a tree in which each vertex is given a unique label. The vertices of a labeled tree on n vertices are typically given the labels 1, 2, …, n. A **recursive tree** is a labeled rooted tree where the vertex labels respect the tree order (*i.e.*, if $u < v$ for two vertices u and v, then the label of u is smaller than the label of v).

An **irreducible** (or *series-reduced*) tree is a tree in which there is no vertex of degree 2.

An **ordered tree** is a tree for which an ordering is specified for the children of each node.

An **n-ary tree** is a tree for which each node which is not a leaf has at the most *n* children. *2-ary trees* (and *3-ary trees* respectively) are sometimes called binary trees (or tertiary trees in case of 3-ary trees)

1.2.1 Properties of Trees

- Every tree is a bipartite graph and a median graph.
- Every connected graph *G* admits a spanning tree, which is a tree that contains every vertex of *G* and whose edges are edges of *G*. Every connected graph even admits a normal spanning tree [17].
- Every finite tree with at least two vertices, say n, has, at least, two *leaves* or vertices of degree 1. The minimal number of leaves corresponds to the path graph and the maximal number (n - 1) corresponds to the star graph.
- For any three vertices in a tree, the three paths among these have at least one vertex in common.

1.2.2 Enumeration

Given *n* labeled vertices, there are n^{n-2} different ways to connect these to make a tree. This result is called Cayley's formula. It can be proved by first showing that the number of trees with *n* vertices of degree $d_1,d_2,\ldots,d_n$ is the multinomial coefficient:

$$\binom{n-2}{d_1-1, d_2-1, \ldots, d_n-1}.$$

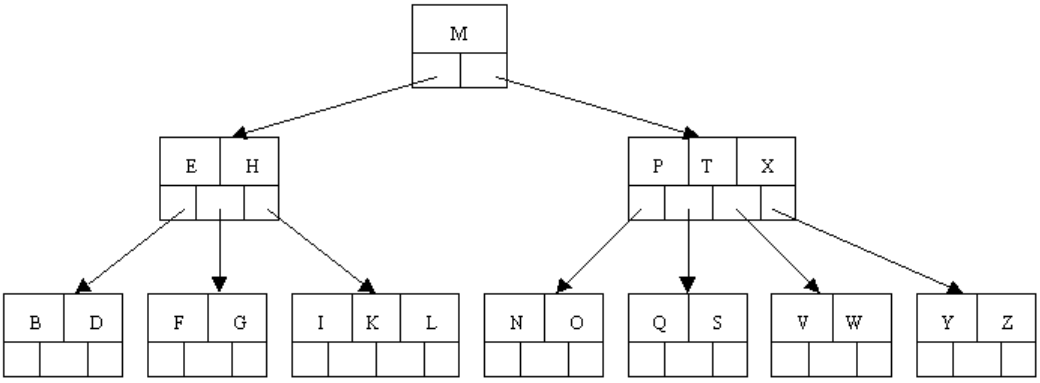
Counting the number of unlabeled trees is a harder problem. No closed formula for the number *t*(*n*) of trees with *n* vertices up to graph isomorphism is known. Otter proved that in 1948.

$$t(n) \sim C\alpha^n n^{-5/2} \quad \text{as } n \to \infty,$$

with $C = 0.53495\ldots$ and $\alpha = 2.95576\ldots$ (here, $f \sim g$ means that $\lim_{n\to\infty} f/g = 1$).

1.2.3 Types of Trees

1.2.3.1 B-Trees



1-2: An example of a B-Tree

A **B-tree** is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in databases and file systems.

A B-tree of order *m* (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at the most *m* children.
- Every node (except root and leaves) has at least $^{m}/_{2}$ children.
- The root has at least two children if it is not a leaf node.
- All leaves appear at the same level, and carry information.
- A non-leaf node with *k* children contains *k*–1 keys

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be

joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

A B-tree is kept balanced by requiring that all external nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and that results in all leaf nodes being one more node further away from the root.

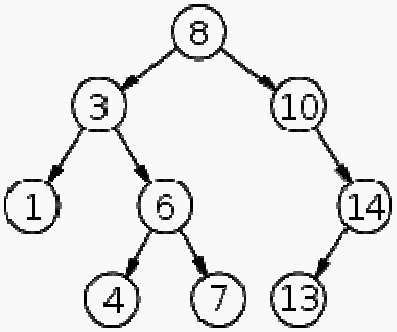
B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or an analogous size in secondary storage. While 2-3 B-trees might be useful in main memory, and are certainly easier to explain, if the node sizes are tuned to the size of a disk block, the result might be a 257-513 B-tree (where the sizes are related to larger powers of 2).

The creators of the B-tree structure, Rudolf Bayer and Ed McCreight, have not explained what, the *B* stands for. Douglas Comer suggests a number of possibilities: "*B*alanced," "*B*road," or "*B*ushy" might apply (since all leaves are at the same level). Others suggest that the "B" stands for Boeing (since the authors worked at Boeing Scientific Research Labs in 1972). Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

1.2.3.2 Binary Tree

A **binary tree** is a tree data structure in which each node has at most two children. Typically the child nodes are called *left* and *right*. Binary trees are commonly used to implement binary search trees and binary heaps.

1.2.3.3 Binary Search Tree



1-3: A Binary Search Tree

A **binary search tree** (**BST**) is a binary tree data structure which has the following properties:

- Each node (item in the tree) has a distinct value.
- Both the left and right subtrees must also be binary search trees.
- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values greater than the node's value.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees can choose to allow or disallow duplicate values, depending on the implementation.

Binary search trees are fundamental data structures used to construct more abstract data structures such as sets, multisets, and associative arrays.

Binary space partitioning (**BSP**) is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of the scene by means of a tree data structure known as a **BSP tree**. In other words, it is a method of breaking up intricately shaped polygons into convex sets, or smaller polygons consisting entirely of non-reflex angles (angles smaller than 180°).

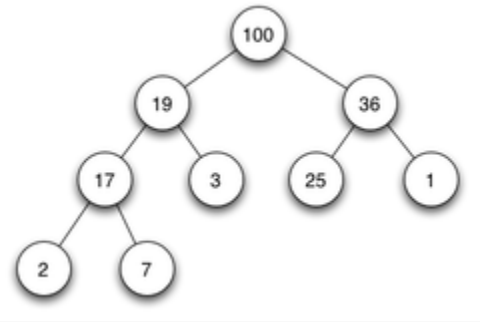
Originally, this approach was proposed in 3-D computer graphics to increase the rendering efficiency. Some other applications include performing geometrical operations with shapes (constructive solid geometry) in CAD, collision detection in robotics and 3-D computer games, and other computer applications that involve handling of complex spatial scenes.

1.2.3.4 B* - Tree

A **B*-tree** is a tree data structure. A variety of B-tree used in the HFS and Reiser4 file systems, which requires non-root nodes to be at least 2/3 full instead of 1/2. To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with the node next to it. When both are full, the two of these are split into three. It also requires the 'leftmost' key never to be used.

The term although, is not in general use today as the implementation was never looked on positively by the computer science community at large; yet most people use "B-tree" generically to refer to all the variations and refinements of the basic data structure.

1.2.3.5 Heaps



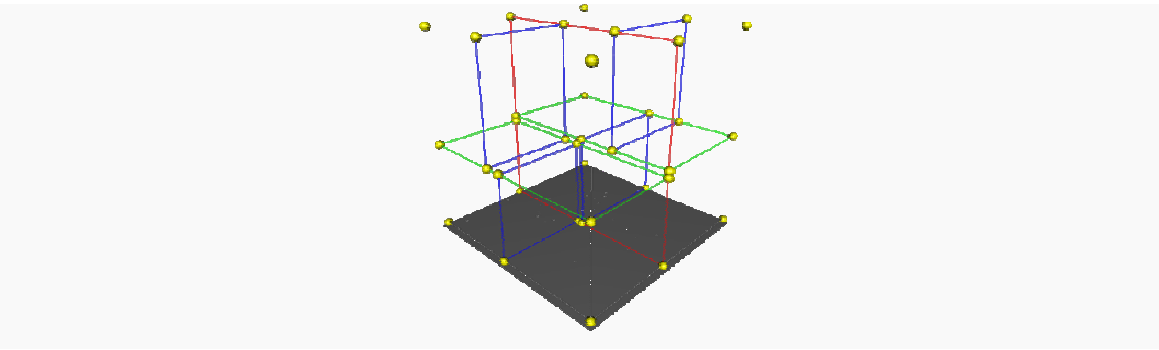
1-4: Example of a full binary Max Heap

A **heap** is a specialized tree-based data structure that satisfies the *heap property:* if *B* is a child node of *A*, then key(*A*) ≥ key(*B*). This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min heap*.) This is why, heaps are used to implement priority queues. The efficiency of heap operations is crucial in several graph algorithms.

The operations commonly performed with a heap are:

- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *insert*: adding a new key to the heap
- *merge*: joining two heaps to form a valid new heap containing all the elements of both.

1.2.3.6 K-D Trees



1-5: A 3-dimensional kd-tree

In computer science, a ***k*d-tree** (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a *k*-dimensional space. *k*d-trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). *k*d-trees are a special case of BSP trees.

1.2.3.7 Cover Tree

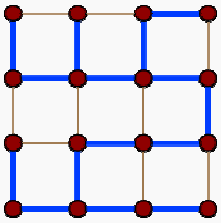
The **cover tree** is a special type of data structure that is specifically designed to facilitate the speed-up of a nearest neighbor search. It was introduced by Alina Beygelzimer, John Langford, and Sham Kakade.

The tree can be thought of as a hierarchy of levels with the top level containing the root point and the bottom level containing every point in the metric space. Each level *C* is

associated with an integer value *i* that decrements by one as the tree is descended. Each level *C* in the cover tree has three important properties:

- *Nesting:* $C_i \subseteq C_{i-1}$
- *Covering:* For every point $p \in C_{i-1}$, there exists a point $q \in C_i$ such that the distance from *p* to *q* is less than or equal to 2^i and exactly one such *q* is a parent of *p*.
- *Separation:* For all points $p, q \in C_i$, the distance from *p* to *q* is greater than 2^i.

1.2.3.8 Spanning Trees



1-6: A Spanning Tree (heavy edges) of a grid graph

In the mathematical field of graph theory, a **spanning tree** *T* of a connected, undirected graph *G* is a tree composed of all the vertices and some (or perhaps all) of the edges of *G*. Informally, a spanning tree of *G* is a selection of edges of *G* that form a tree *spanning* every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of *G* must belong to *T*.

1.2.3.9 Exponential Tree

An **exponential tree** is almost identical to a binary search tree, with the exception that the dimension of the tree is not the same at all levels. In a normal binary search tree, each node has a dimension (*d*) of 1, and has 2^d children. In an exponential tree, the dimension equals the depth of the node, with the root node having a $d = 1$. So, the second level can hold two nodes, the third can hold eight nodes, the fourth, 64 nodes, and so on.

1.2.3.10 K-ary Tree

A **k-ary tree** is a rooted tree in which each node has no more than *k* children. It is also sometimes known as a **k-way tree**, an **N-ary tree**, or an **M-ary tree**.

A binary tree is the special case where $k=2$.

A **full k-ary tree** is a k-ary tree where each node has either *0* or *k* children.

For a k-ary tree with height *h*, the upper bound for the maximum number of leaves is k^h.

Chapter 2 : Spanning Trees

In the mathematical field of graph theory, a **spanning tree** *T* of a connected, undirected graph *G* is a tree composed of all the vertices and some (or perhaps all) of the edges of *G*. Informally a spanning tree of G is a selection of edges of G that form a tree *spanning* every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of *G* must belong to *T* [18].

A spanning tree of a connected graph *G* can also be defined as a maximal set of edges of *G* that contains no cycle, or as a minimal set of edges that connects all vertices [17].

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components [17].

The concept of minimum spanning tree can be understood through an example [17]: A TV company whishes to provide cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because these are longer, or require the cable to be buried deeper. These paths would be represented by edges with larger weights. A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost.

In certain fields of graph theory it is often useful to find a minimum spanning tree of a weighted graph. Other optimization problems on spanning trees have also been studied,

including the maximum spanning tree, the minimum tree that spans at least k vertices, the minimum spanning tree with at most k edges per vertex (MDST), the spanning tree with the largest number of leaves (closely related to the smallest connected dominating set), the spanning tree with the fewest leaves (closely related to the Hamiltonian path problem), the minimum diameter spanning tree, and the minimum dilation spanning tree.

2.1. Fundamental Cycle

Adding just one edge to a spanning tree will create a cycle; such a cycle is called a **fundamental cycle**. There is a distinct fundamental cycle for each edge; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with V vertices, any spanning tree will have V-1 edges, and thus, a graph of E edges will have E-V+1 fundamental cycles. For any given spanning tree, these cycles form a basis for the cycle space [18].

Similar to the concept of a fundamental cycle is the notion of a **fundamental cutset**. By deleting just one edge of the spanning tree, the vertices are partitioned into two disjoint sets. The fundamental cutset is defined as the set of edges that must be removed from the graph G to accomplish the same partition [18]. Thus, there are precisely V-1 fundamental cutsets for the graph: one for each edge of the spanning tree.

The duality between fundamental cutsets and fundamental cycles is established by noting that cycle edges not in the spanning can only appear in the cutsets of the other edges in the cycle; and *vice versa*: edges in a cutset can only appear in those cycles not containing the edge corresponding to the cutset.

2.2. Spanning Forest

A **spanning forest** is a type of subgraph that generalizes the concept of a spanning tree. However, there are two definitions in common use [18]. One is that a spanning forest is a subgraph that consists of a spanning tree in each connected component of a graph. (Equivalently, it is a maximal cycle-free subgraph.) This definition is common in computer science and optimization. It is also the definition used when discussing

minimum spanning forests, the generalization to disconnected graphs of minimum spanning trees. Another definition, common in graph theory, is that a spanning forest is any subgraph that is both a forest (contains no cycles) and a spanning tree (includes every vertex).

2.3. Counting spanning Trees

The number *t(G)* of spanning trees of a connected graph is an important invariant. In some cases, it is easy to calculate *t(G)* directly. It is also widely used in data structures in different computer languages. For example, if *G* is itself a tree, then *t(G)=1*, while if *G* is the cycle graph C_n with *n* vertices, then *t(G)=n* [18]. For any graph *G*, the number *t(G)* can be calculated using Kirchhoff's matrix-tree theorem .

Cayley's formula is a formula for finding the number of spanning trees in the complete graph K_n with *n* vertices. The formula states that $t(K_n) = n^{n-2}$. Another way of stating Cayley's formula is that there are exactly n^{n-2} labeled trees with *n* vertices. Cayley's formula can be proved using Kirchhoff's matrix-tree theorem or via the Prüfer code.

If *G* is the complete bipartite graph $K_{p,q}$, then $t(G) = p^{q-1}q^{p-1}$, while if *G* is the *n*-dimensional hypercube graph Q_n, then $t(G) = 2^{2^n-n-1}\prod_{k=2}^{n} k^{\binom{n}{k}}$. These formulae are also consequences of the matrix-tree theorem [18].

If *G* is a multigraph and *e* is an edge of *G*, then the number *t(G)* of spanning trees of *G* satisfies the *deletion-contraction recurrence t(G)=t(G-e)+t(G/e)*, where *G-e* is the multi-graph obtained by deleting *e* and *G/e* is the contraction of *G* by *e*, where multiple edges arising from this contraction are not deleted.

2.4. Uniform Spanning Trees

A spanning tree chosen randomly from among all the spanning trees with equal probability is called a uniform spanning tree (UST). This model has been extensively researched in probability and mathematical physics.

2.5. Properties of Spanning Trees

2.5.1. Possible Multiplicity

There may be several minimum spanning trees of the same weight. In particular, if all weights are the same, every spanning tree is minimum.

2.5.2. Uniqueness

If each edge has a distinct weight then there will only be one, unique minimum spanning tree. This fact can be proved by induction or contradiction. This is true in many realistic situations, such as the cable TV company example above, where it is unlikely that any two paths have *exactly* the same cost. This generalizes to spanning forests as well.

2.5.3. Minimum-cost Sub-graph

If the weights are non-negative, then a minimum spanning tree is in fact the minimum-cost subgraph connecting all vertices; since subgraphs containing cycles necessarily have more total weight.

2.5.4. Cycle Property

For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of other edges of C, then this edge cannot belong to an MST. Indeed, assume the contrary, i.e., e belongs to an MST T1. If we delete it, T1 will be broken into two subtrees with the two ends of e in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge f of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree T2 with weight less than that of T1, because the weight of f is less than the weight of e [19].

2.5.5. Cut Property

For any cut C in the graph, if the weight of an edge e of C is smaller than the weights of other edges of C, then this edge belongs to all MST's of the graph. Indeed, assume the contrary, i.e., *e* does not belong to an MST T1, then adding *e* to T1 will produce a cycle, which must have another edge *e2* from T1 in the cut *C*. Replacing *e2* with *e* would produce a tree T1 of smaller weight [19].

2.6. Time Complexities of finding MSTs

The first algorithm for finding a minimum spanning tree was developed by Czech scientist Otakar Borůvka in 1926. Its purpose was an efficient electrical coverage of Moravia. There are now two algorithms commonly used: Prim's algorithm and Kruskal's algorithm. All three are greedy algorithms that run in polynomial time. Another greedy algorithm not as commonly used is the reverse-delete algorithm, which is the reverse of Kruskal's algorithm.

The fastest minimum spanning tree algorithm to date was developed by Bernard Chazelle, which is based on the Soft Heap, an approximate priority queue. Its running time is $O(e\ \alpha(e,v))$, where e is the number of edges, v is the number of vertices and α is the classical functional inverse of the Ackermann function. The function α grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4; thus Chazelle's algorithm takes very close to linear time [21].

There is clearly a linear lower bound, since we must, at least, examine all the weights. If the edge weights are integers with a bounded bit length, then deterministic algorithms are known with linear running time. For general weights, there are randomized algorithms whose *expected* running time is linear.

Whether there exists a deterministic algorithm with linear running time for general weights, is still an open question. However, Seth Petie and Vijaya Ramachandran have found a provably optimal deterministic minimum spanning tree algorithm, the computational complexity of which is unknown.

More recently, research has focused on solving the minimum spanning tree problem in a highly parallelized manner. With a linear number of processors it is possible to solve the problem in $O(\log n)$ time. A 2003 paper "Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs" by David A. Bader and Guojing Cong demonstrates a pragmatic algorithm that can compute MSTs 5 times faster on 8 processors than an optimized sequential algorithm. Typically, parallel algorithms are based on Boruvka's algorithm; Prim's and especially Kruskal's algorithm do not scale as well to additional processors.

Other specialized algorithms have been designed for computing minimum spanning trees of a graph so large that most of it must be stored on disk at all times. These *external storage* algorithms, for example as described in [19] can operate as little as 2 to 5 times slower than a traditional in-memory algorithm. They claim that "massive minimum spanning tree problems filling several hard disks can be solved overnight on a PC." They rely on efficient external storage sorting algorithms and on graph contraction techniques for reducing the graph size efficiently.

2.7. Applications of Minimum Spanning Trees

The standard application is a problem like phone network design. Suppose, there is a business with several offices; and leased phone lines to connect these up with each other are to be set up; and the phone company charges different amounts of money to connect different pairs of cities. We want a set of lines that connects all the offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree we can always remove some edges and save money.

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if we have a path visiting all points exactly once, it is a special kind of tree. If we have a path visiting some vertices more than once, we can always drop some edges to get a tree. So, in general the MST weight is less than the TSP weight, because it is a minimization over a strictly larger set.

On the other hand, if we draw a path tracing around the minimum spanning tree, we trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

2.8. Types

2.8.1. Minimum *K*-Spanning Tree

- It is a *k*-spanning tree, i.e., a subtree *T* of *G* of at least *k* nodes.
- It is represented by the graph $G = (V, E)$, an integer $k \leq n$, and a weight function $w : E \rightarrow N$.

2.8.2. Minimum Degree Spanning Tree

- It is a spanning tree for *G* with the degree of the tree being the maximum degree of the spanning graph.
- It is represented by the graph $G = (V, E)$, and the integer representing the maximum degree of the spanning graph.

2.8.3. Minimum Geometric 3-Degree Spanning Tree

- It is a spanning tree *T* for *P* in which no vertex has degree larger than 3.
- It is represented by the set $P \subseteq Z \times Z$ of points in the plane.
- The total weight of the spanning tree, is represented by $\sum_{(u,v)\in T} d(u, v)$, where *d(u,v)* is the Euclidean distance between *u* and *v*.

2.8.4. Maximum Leaf Spanning Tree

- It is a spanning tree for *G*, in which the number of leaves of the spanning tree is maximum.
- It is represented by graph $G = (V, E)$.
- Other problems which aim at finding spanning trees that maximize a single objective function have been considered. In particular, the problems of finding a spanning tree that has maximum diameter, or maximum height with respect to a specified root, and the problems of finding a spanning tree that has maximum sum

of the distances between all pairs of vertices, or maximum sum of the distances from a specified root are also common.

2.8.5. Maximum Minimum Metric *K*-Spanning Tree

- It is a spanning tree of a graph in which a subset $V' \subseteq V$ of vertices is used such that $|V'| = k$.
- It is represented by a graph $G = (V, E)$, length $l(e) \in N$ for each $e \in E$ satisfying the triangle inequality.
- Cost of the minimum spanning tree of the subgraph is induced by *V'*.
- Also called *Maximum Remote Minimum Spanning Tree*.

2.8.6. Minimum Diameter Spanning Subgraph

- It is a spanning subgraph $E' \subseteq E$ for *G* such that the sum of the weights of the edges in *E'* does not exceed *B*.
- It is represented by a graph $G = (V, E)$, weight $w(e) \in Z^+$ and length $l(e) \in N$ for each $e \in E$, positive integer *B*.
- B represents the diameter of the spanning subgraph.

2.8.7. Minimum Communication Cost Spanning Tree

- A spanning tree for *G* where the weighted sum over all pairs of vertices of the cost of the path between the pair in *T*, is represented by $\sum_{u,v \in V} W(u,v) \cdot r(\{u,v\})$, where *W(u,v)* denotes the sum of the weights of the edges on the path joining *u* and *v* in *T*.
- It is represented by the complete graph $G = (V, E)$, weight $w(e) \in N$ for each $e \in E$, requirement $r(\{u,v\}) \in N$ for each pair of vertices from *V*.

- It is approximable within $O(\log^2 |V|)$ if the weight function satisfies the triangle inequality.

2.8.8. Minimum Steiner Tree

- A Steiner tree, i.e., a subtree of *G* that includes all the vertices in *S* and the sum of the weights of the edges in the subtree.
- It is represented by the complete graph $G = (V, E)$, a metric given by edge weights $s : E \to N$ and a subset $S \subseteq V$ of required vertices.

2.8.9. Minimum Geometric Steiner Tree

- It is a sub-graph in which the total weight of the minimum spanning tree is the same as for the vertex set $P \cup Q$, where the weight of an edge $\langle (x_1, y_1), (x_2, y_2) \rangle$ is the discretized Euclidean length

$$\left\lceil \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \right\rceil .$$

- It is represented by a set $P \subseteq Z \times Z$ of points in the plane and a finite set of Steiner points, i.e. $Q \subseteq Z \times Z$.

2.8.10. Minimum Generalized Steiner Network

- It is a Steiner network over *G* that satisfies all the requirements and obeys all the capacities, i.e., a function $f : E \to N$ such that, for each edge *e*, $f(e) \leq d(e)$ and, for any pair of nodes *i* and *j*, the number of edge disjoint paths between *i* and *j* is at least *r(i,j)* where, for each edge *e*, *f(e)* copies of *e* are available.

- It is represented by the graph $G = (V, E)$, a weight function $w : E \to N$, a capacity function $c : E \to N$, and a requirement function $r : V \times V \to N$.
- The cost of the network is $\sum_{e \in E} w(e) f(e)$.
- Also called *Minimum Survivable Network*..

2.8.11. Minimum Routing Tree Congestion

- A routing tree *T* for *G*, i.e., a tree *T* in which each internal vertex has degree 3 and the leaves correspond to vertices of *G*.
- It is represented by the graph $G = (V, E)$ and a weight function $w : E \to N$.
- The congestion of the routing tree, i.e., the maximum, for any edge *e*, of

$$\sum_{(u,v) \in E, u \in S, v \notin S} w(u, v)$$

where *S* is one of the two connected components obtained by deleting *e* from *T*.

2.8.12. Maximum Minimum Spanning Tree Deleting *K* Edges

- It is a subset $E' \subseteq E$ of *k* edges and a minimum spanning tree *T* in the graph $(V, E - E')$.
- It is represented by a graph $G = (V, E)$, a weight function $w : E \to N$.

2.8.13. Minimum Upgrading Spanning Tree

- It is a spanning tree made from an upgrading set $W \subseteq V$ of vertices such that the weight of a minimum spanning tree in *G* with respect to edge weights given by d_W is bounded by *D*. Here, d_W denotes the edge weight function resulting

from the upgrade of the vertices in *W*, i.e., $d_W(u,v) = d_i(u,v)$ where $|W \cap \{u,v\}| = i$.

- It is represented by the graph *G=(V,E)*, three edge weight functions $d_2(e) \leq d_1(e) \leq d_0(e)$ (for each $e \in E$), where $d_i(e)$ denotes the weight of edge *e* if *i* of its endpoints are ``upgraded'', vertex upgrade costs *c(v)* (for each $v \in V$), a threshold value *D* for the weight of a minimum spanning tree.
- The cost of the upgrading set is represented by: $c(W) = \sum_{v \in W} c(v)$.
- Approximable within $O(\log |V|)$ if the difference of the largest edge weight $D_0 = \max\{d_0(e) : e \in E\}$ and the smallest edge weight $D_2 = \min\{d_2(e) : e \in E\}$ is bounded by a polynomial in $|V|$
- Variation in which the upgrading set must be chosen such that the upgraded graph contains a spanning tree in which no edge has weight greater than *D* is approximable within $2 \ln |V|$. In this case no additional assumptions about the edge weights are necessary.

2.9. Algorithms for finding Minimum Spanning Trees

2.9.1. Kruskal's algorithm

Kruskal's algorithm is the easiest to understand and probably the best one for solving problems by hand. **Kruskal**'s algorithm for the minimum spanning tree problem begins with many disjoint spanning trees, a spanning forest. It repeatedly joins two trees together until a spanning tree of the entire given graph remains [20].

Kruskal's algorithm:

sort the edges of G in increasing order by length

keep a subgraph S of G, initially empty

for each edge e in sorted order

if the endpoints of e are disconnected in S

add e to S

return S

Note that, whenever an edge (u,v) is added, it is always the smallest connecting the part of S reachable from u with the rest of G. In other words, it can described as:

--G = <V,E>

P := {{v_1}, ..., {v_n}} --partition V into singleton trees

E' := {}

loop |V|-1 times

--Inv: E' contains only edges of a min' span' tree for each S in P &

-- each S in P represents a subtree of a minimum spanning tree of G

find shortest edge e joining different subsets S1 and S2 in P

E' += {e}

P := P - {S1,S2} + {S1 union S2}

end loop

This algorithm is known as a *greedy algorithm*; because it chooses, at each step, the cheapest edge to add to S. Greedy algorithms however, must be used with care to solve other problems, since these sometimes don't work. E.g. if you want to find the shortest path from a to b, it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property.

The line testing whether two endpoints are disconnected, looks like it should be slow (linear time per iteration, or O(mn) total). But actually, there are some complicated data

structures that let us perform each test in almost constant time; this is known as the *union-find* problem. The slowest part turns out to be the sorting step, which takes O(m log n) time. The time complexity of Kruskal's algorithm hinges on finding the shortest edge to join two subtrees and on the joining itself. A priority queue can be used to organize the edges. A heap is a suitable implementation. The priority queue takes O(|E|log|E|) time to create and O(log|E|) time to find the shortest edge while maintaining the priority queue. The latter is done |V|-1 times. The joining of two subtrees can be done in O(|V|log|V|) total time over all the iterations. Thus Kruskal's algorithm takes O(|E|log|E|) time overall [20]. This is better than Prim's algorithm for sparse graphs for which $|E|<<|V|^2$.

The forest is represented by a partition of the vertices of the graph. Each partial tree spans a subset of the vertices. An array gives the size and first member of each subset. A second array gives the subset of each member and links the members of a subset together.

When two subsets are joined, the members of one subset can be transferred to the other. It is more efficient to transfer the members of the smaller subset to the larger as there are fewer of these. The smaller subset S is joined to a subset at least as large as itself and the resulting subset is at least twice as large as S. Thus, each vertex is transferred at most log|V| times. This operation takes place |V|-1 times. The time taken, over all the join operations, is O(|V|log|V|), so this is not a dominant part of the time complexity of Kruskal's algorithm [20].

Kruskal's algorithm certainly leads to a spanning tree T but it is necessary to prove that T is minimal. The invariant shows that this is so: The invariant is certainly true on the initial iteration. In the body of the loop, two partial minimum spanning trees T1 and T2 are joined by an edge `e'. The vertices in T1 and T2 must be connected somehow in a final minimum spanning tree T'. Suppose these could be connected more cheaply in T' than in T. Add e to T'. This would create a cycle but it could be broken by removing an edge from T to T'-T. This would leave the vertices in T1 and T2 connected but at a lower cost than in T' because e is chosen as the cheapest satisfactory edge. Therefore T1, T2 and e form a minimum spanning subtree which must be part of a minimum spanning tree

of G [20]. The algorithm continues to join subtrees until a single minimum cost tree remains, spanning all vertices.

2.9.2. Prim's algorithm

Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time. **Prim**'s algorithm for the minimum spanning tree problem follows the strategy of beginning with a small tree, i.e. <{v_1},{ }>, and growing it until it includes all vertices in the given graph. Initially, the tree contains just an arbitrary starting node v_1. At each stage a vertex not yet in the tree but closest to (some vertex that is in) the tree is found. This closest vertex is added to the tree. Finding the vertex allows us to improve our knowledge of the distances of the remaining vertices to the tree. A set `done' contains the vertices in the tree [20].

```
--Graph G = <V, E>
done := {v1}   --initial Tree is <{v1},{}>
for vertex i in V-{v1}
  T[i] := E[1,i]     --direct edges (possibly "missing")
end for
loop |V|-1 times
--Inv: {T[v]|v in done} represents a min' spanning Tree
--     of the nodes in done and
--     {T[u]|u not in done} contains the shortest known
--     distances from the (sub-)spanning Tree to
--     such vertices u.
 find closest vertex to (sub-)spanning Tree in V - done
 done +:= {closest}
 add closest & edge T[closest] to (sub-)spanning Tree
```

```
 for vertex j in V - done
   T[j] := min(T[j],   --update knowledge on paths,
             E[closest,j])      --perhaps better?
 end for
end loop
```

Each edge added is the smallest connecting T to G-T.

Again, the loop has a slow step in it. But again, some data structures can be used to speed this up. The idea is to use a heap to remember, for each vertex, the smallest edge connecting T with that vertex.

```
   Prim with heaps:
   make a heap of values (vertex,edge,weight(edge))
      initially (v,-,infinity) for each vertex
      let tree T be empty
   while (T has fewer than n vertices)
   {
      let (v,e,weight(e)) have the smallest weight in the heap
      remove (v,e,weight(e)) from the heap
      add v and e to T
      for each edge f=(u,v)
      if u is not already in T
         find value (u,g,weight(g)) in heap
         if weight(f) < weight(g)
         replace (u,g,weight(g)) with (u,f,weight(f))
   }
```

A total of ‘n’ steps are performed in which the smallest element in the heap is removed, and at most 2m steps are performed in which an edge f=(u,v) is examined. For each of those steps, a value on the heap might be replaced, reducing its weight. The weight of an element of a binary heap can be reduced in O(log n) time. Alternatively, by using a more complicated data structure known as a Fibonacci heap, the weight of an element can be reduced in constant time. The result is a total time bound of O(m + n log n) [20].

Prim's algorithm is very similar to Dijkstra's single-source shortest path algorithm and indeed the given version of the former was created by simple edits to the latter. The principal difference is that the criterion for choosing a new vertex is proximity to the tree rather than to a source. Prim's algorithm also clearly takes O($|V|^2$) time [20].

Prim's algorithm is correct because at each stage it has built a minimum spanning tree over those vertices in the set `done' which eventually contains all the vertices: The condition is trivially true initially. Consider the addition of a "closest" vertex `v' by an edge `e' to the partial spanning tree T at some intermediate stage. Vertex v must be in the final minimum spanning tree T' of all G. Suppose v could be connected into T' by some other path, not requiring e, for a smaller total cost. Now add e to T'. This would create a cycle through part of T. The cycle could be broken by deleting an edge out of T into the rest of T' of higher weight than e, because e is chosen as the lowest cost edge out of T to a vertex that has not already been added. Thus T' could not be a minimum spanning tree of G, i.e. a contradiction, so the supposition is false.

2.9.3. Boruvka's algorithm

This is probably the easiest algorithm for computer implementation since it does not require any complicated data structures. But it is relatively difficult to describe. The idea is to perform steps like Prim's algorithm, in parallel all over the graph at the same time [20]

Boruvka's algorithm:

make a list L of n trees, each a single vertex

while (L has more than one tree)

for each T in L, find the smallest edge connecting T to G-T

add all those edges to the MST

(causing pairs of trees in L to merge)

As in case of Prim's algorithm, each edge added must be part of the MST, so it must be all right to add these all at once.

This is similar to merge sort. Each pass reduces the number of trees by a factor of two, so there are O(log n) passes. Each pass takes time O(m) (first figure out which tree each vertex is in, then for each edge test whether it connects two trees and is better than the ones seen before for the trees on either endpoint) so the total is O(m log n).

2.9.4. A hybrid algorithm

This is not a separate algorithm, but a combination of two of the classical algorithms which in turn does better than either one alone. The idea is to do O(log log n) passes of Boruvka's algorithm, then switch to Prim's algorithm. Prim's algorithm then builds one large tree by connecting it with the small trees in the list L built by Boruvka's algorithm, keeping a heap which stores, for each tree in L, the best edge that can be used to connect it to the large tree. Alternatively, the trees found by Boruvka's algorithm can be collapsed into "supervertices" and Prim's algorithm run on the resulting smaller graph. The point is that this reduces the number of remove min operations in the heap used by Prim's algorithm, to equal the number of trees left in L after Boruvka's algorithm, which is O(n / log n) [20]. The psuedocode of the algorithm is described as under:

make a list L of n trees {O(log log n) passes of Boruvka's algorithm}

For each T in L

  Keep a heap which stores the best edge which can be used for connecting

Build one large tree by connecting the small trees in L {Prim's algorithm}

It takes O(m log log n) for the first part, O(m + (n/log n) log n) = O(m + n) for the second, so O(m log log n) total time is taken [20].

A graph often contains redundancy in that there can be multiple paths between two vertices. This redundancy may be desirable, for example to offer alternative routes in the case of breakdown or overloading of an edge (road, connection, phone line) in a network. However, the cheapest sub-network that connects the vertices of a given graph is often required. This must, in fact, be an *unrooted tree*, because there is only one path between any two vertices in a tree. If there is a cycle then at least one edge can be removed. The total cost or weight of a tree is the sum of the weights of the edges in the tree. We assume that the weight of every edge is greater than zero. Given a connected, undirected graph G=<V,E>, the *minimum spanning tree problem* is to find a tree T=<V,E'> such that E' is subset of E and the cost of T is minimal.

Note that a minimum spanning tree is not necessarily unique. Recall that a tree over |V| vertices contains |V|-1 edges. A tree can be represented by an array of this many edges.

Chapter 3 : Design and Implementation of the System

3.1. Background

Pattern recognition is a subtopic of machine learning. It is the act of taking in raw data and taking an action based on the category of the data.

Pattern recognition essentially enables the classification of objects into groups by learning from a small sample of objects [1]. The objects are represented by a number of features that describe those objects. These features are then used to classify the objects into one of the classes. Often, there are a large number of features and it is not feasible to use all of these to classify the objects. This could be due to the cost of measuring and recording the features, or the computational cost. Increasing the number of features used for classification, generally increases the classification process exponentially. Also some features may be redundant or noisy which may result in classification errors.

3.1.1. Kinds of Learning

Supervised Learning is that type of learning in which a set of example input/output pairs are given and the problem is to find a rule that does a good job of predicting the output associated with a new unseen input.

Unsupervised Learning (also called **Clustering**) is the type of machine learning in which a set of examples is given but no labeling of those examples is given. The problem is to group those examples into "natural" clusters.

Reinforcement Learning is that type in which an agent interacting with the world makes observations, takes actions and is rewarded or punished. Based on that, it learns to choose actions in such a way, so as to obtain maximum reward.

Pattern recognition aims to classify data (patterns) based either on *a priori* knowledge or statistical information extracted from the patterns. The patterns to be classified are usually groups of measurements or observations, defining points in an

appropriate multidimensional space. This is in contrast to pattern matching, where the pattern is rigidly specified.

A complete pattern recognition system consists of a sensor that gathers the observations to be classified or described, a feature extraction mechanism that computes numeric or symbolic information from the observations, and a classification or description scheme that does the actual job of classifying or describing observations, relying on the extracted features.

The classification or description scheme is usually based on the availability of a set of patterns that have already been classified or described [1]. This set of patterns is termed the training set, and the resulting learning strategy is characterized as supervised learning. Learning can also be unsupervised, in the sense that the system is not given an *a priori* labeling of patterns, instead, it itself establishes the classes based on the statistical regularities of the patterns by making clusters and assigning samples to different clusters.

The classification or description scheme usually uses one of the following approaches: statistical (or decision theoretic) or syntactic (or structural). Statistical pattern recognition is based on statistical characterizations of patterns, assuming that the patterns are generated by a probabilistic system. Syntactical (or structural) pattern recognition is based on the structural interrelationships of features. A wide range of algorithms can be applied for pattern recognition, from very simple Bayesian classifiers to much more powerful neural networks [1].

An intriguing problem in pattern recognition is the relationship between the problem to be solved (data to be classified) and the performance of various pattern recognition algorithms (classifiers). Van der Walt and Barnard investigated very specific artificial data sets to determine conditions under which certain classifiers perform better and worse than others.

Pattern recognition is more complex when templates are used to generate variants. For example, in English, sentences often follow the "N-VP" (noun - verb phrase) pattern, but some knowledge of the English language is required to detect the pattern. Pattern recognition is studied in many fields, including psychology, ethology, cognitive science and computer science.

Holographic associative memory is another type of pattern matching scheme where a target small patterns can be searched from a large set of learned patterns based on cognitive meta-weight.

Within medical science, pattern recognition is the basis for computer-aided diagnosis (CAD) systems. CAD describes a procedure that supports the doctor's interpretations and findings.

Typical applications are automatic speech recognition, classification of text into several categories (e.g. spam/non-spam email messages), the automatic recognition of handwritten postal codes on postal envelopes, or the automatic recognition of images of human faces. The last two examples form the subtopic image analysis of pattern recognition that deals with digital images as input to pattern recognition systems.

In pattern recognition, objects and samples are described by different feature values. These feature values are the properties of the objects and are the basis on which the objects can be classified. However, in real world, objects may be described by hundreds and thousands of features. By taking into consideration each and every feature, not only is the time to train the system increased but also some feature values may be redundant and may not provide any useful information for classification.

For example, consider the case where we have to classify two types of fruits: red apples and strawberries. These both may be represented by many features. If we choose the feature ‘color’ for classification, it is a redundant feature which does not provide us any information and does not aide us in classifying the fruits since in this case; the feature value for all samples would be ‘red’.

A subsequent problem of pattern recognition is, therefore, finding a reduced set of features to enable the classification of objects to be done efficiently and accurately. This not only reduces the training time but also results in better accuracy.

Tou and Gonzalez (1974) believe that feature selection plays a central role in pattern recognition, and is one of the most difficult tasks. This research consequently focuses on performing feature subset selection. A criterion function will be developed which

effectively rates feature subsets by building minimum cost spanning trees across the sample data-sets.

3.1.2. Pattern Recognition

According to Ray (2005c), pattern recognition can be divided into three stages: pattern representation, feature selection and classification.

3.1.2.1. Pattern Representation

The pattern representation stage, Ray (2005c) explains, is concerned with measuring the features from the objects and recording the data so that a computer is able to process it.

For example, if we were classifying apples and strawberries, then this stage may involve taking different weight and size measurements of the fruits. This stage essentially develops the data-sets.

3.1.2.2. Feature Set Reduction

A problem with pattern recognition, explains Friedman and Kandel (1999), is that often it is not possible to use optimal features. An example by Tou and Gonzalez (1974) is character recognition. This is the problem of interpreting human written characters. The best way may be to record the order of the strokes for each character and where each stroke begins and ends. For various reasons this may not be feasible, so instead we may record other features. A solution by Ray and Turner (1992) plots the characters onto a grid and counts the number of times pixels occur in each column and each row. This will not be as effective, but it can be easily measured and recorded. One can imagine that some rows and columns will be less useful than others, for example the borders, since most characters would have the same value of zero. We may then want to discard the less useful features, since these do not provide extra information about the object. The importance of removing redundant features is further highlighted by a comment from Jain et al. (2000), stating that using a large number of redundant features can make two different patterns to appear similar.

The process of discarding features is called feature subset selection. Alternatively, we can combine features mathematically and produce a new reduced feature set. This is called

feature extraction. According to Jain and Zongker (1997), there has been an increase of interest in this area due to increases in interest in areas, such as data mining.

The aim of feature set reduction, according to Jain and Dubes (1988) and Jain et al.

(2000), is to decrease the cost of recognition and increase the classification accuracy (particularly when the number of samples is small in relation to the number of features and classes of the dataset). An important aspect of feature set reduction is speed. Even though this step is typically only ever run once, speed is still an issue.

As explained, there are two methods of reducing the number of features, Jain et al. (2000) states that often both are used, feature extraction first to extract the key data from the feature set, and then feature subset selection to discard features which provide little information about the classes.

3.1.2.3. Feature Subset Selection

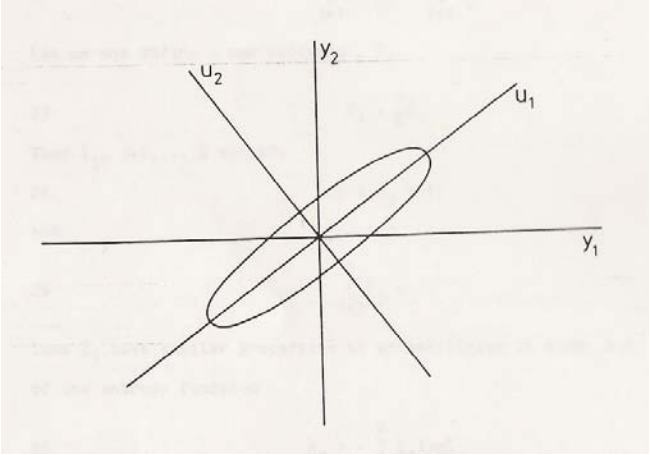
In feature subset selection, subset of the original features is selected to use for the classification process. Jain and Zongker (1997) explain feature subset selection as essentially trying to find a feature set X, for which $X \subset Y$, where Y is the entire feature set. The subset X is chosen such that it optimizes some criterion C. Feature subset selection can, therefore, be seen as two separate processes:

1. Obtaining a criterion function, “C”, which effectively rates feature subsets. This is the main issue in feature set reduction, according to Jain et al. (2000), and it is the focus of this research.

2. Feature subset searching. This step is necessary, because it is not feasible to run the proposed algorithm exhaustively on every feature subset possible (Jain and Zongker, 1997; Narendra and Fukunaga, 1977; Clausen, 1999). If, for example, we have 10 features to select from, then the number of different possible subsets are: ${}^{10}C_1 + {}^{10}C_2 + {}^{10}C_3 \ldots + {}^{10}C_{10} = 1023$. This process increases combinatorially. If we have 20 features, then there are over 1 million different subsets. This step involves searching possible subsets, with the goal of optimizing the criterion function “C”.

3.1.2.4. Feature Extraction

Feature extraction, on the other hand, transforms the original set of features into a smaller set. Ray (2005b) explains that feature extraction takes an original set of features, Y = (y_1....y_n) and transforms these into a smaller set X = (x_1....x_m), where m < n. Note that this is different from feature subset selection, in that the transformed features are not a subset of the original, but rather a product or function of these. An example of a commonly used feature extraction method is outlined below [1].



3-1: Feature Extraction by the Karhunen-Loeve transform

- **Karhunen-Loeve Transform**

The key idea in the Karhunen-Loeve transform is, according to Devijver and Kittler (1982), to compress the information contained in the original set Y to set X. The Karnhunen- Loeve expansion works by de-correlating the features of Y, and then removing the features which do not contain significant information, which are those which have a smaller variance. Figure 3-1 from Devijver and Kittler (1982) shows the effect of the Karhunen-Loeve transformation. The original features (y1, y2) are shown, and can be seen to be highly correlated. The transformed features (u1, u2) are not correlated, and it is clear that u1 provides significantly more discriminatory information than u2; therefore u2 can be discarded without much loss in information.

- **Feature Set Reduction Speed**

An obvious question may be that as feature set reduction is typically done off line, the execution time of the process is not particularly important. As Jain and Zongker (1997) explain, this is valid to a point; however the process will need to be run at least once.

The execution time becomes a problem when processing large data sets with hundreds and thousands of features. The execution time of the algorithm is then a crucial issue, as it may not be feasible to run some algorithms even once on large data sets.

3.1.2.5. Classification

This is the final stage, where the reduced feature subset and, in general, knowledge learnt from the sample data is used to classify objects according to their classes. Here, the Bayesian classifier is introduced, which is the optimal classification method, and several other commonly used methods are discussed.

- ***Bayesian Classifier***

The Bayesian classifier is the optimal classification method. It uses the a'priori probabilities of the classes (which is the initial probabilities of the class, for example we may know that 60% of objects belong to class 1 and 40% to class 2) and the probability of an input pattern occurring in a given class to obtain the probability of a class occurring given an input pattern. This is expressed in Ray (2005a) as:

$$P(W_i|X) = \frac{P(X) * P(X|W_i)}{P(X)}$$

Where W_i is class i, and X is the input pattern. Note that P(X) does not need to be calculated, as it bears the same value for all of the classes. The equation can then be calculated as:

$$P(W_i|X) = P(X) * P(X|W_i)$$

Unfortunately, according to Ray and Turner (1992), this method is, at best, difficult to compute, if not possible at all. To solve this, other classification methods have been introduced to attempt to estimate this classification process.

- ***Euclidean Distance Classifier***

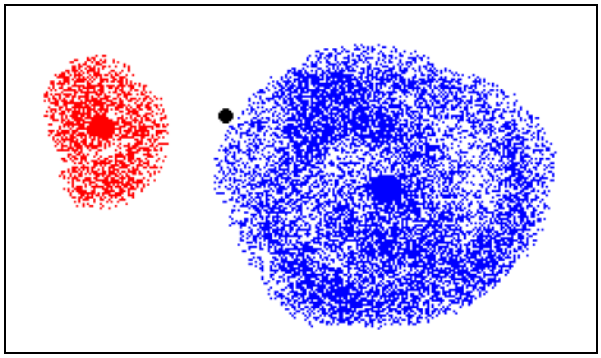
For an input pattern x, Friedman and Kandel (1999) explains, this classifier will classify x according the class mean it is closest to. The mean of each class is found by taking its average feature vector, and the euclidean distance from x to each class mean is calculated as:

$$d_i^2 = (x - \mu_i)'(x - \mu_i)$$

Where μ_i is the mean feature vector of class i and x is the feature vector of input pattern x. The input pattern x is then assigned to the class i if $d_i < d_j$ for all classes j = 1...n.

Since we are not interested in the actual values of d_x, just whether the distance from one class is smaller than another, we can use d^2_x and obtain the same result, without the expensive square root operation.

This is a quite a simple method, however it does have its drawbacks. For example, consider figure 3-2, the case of two classes, where one class has a much larger standard deviation than the other. This classifier will assign the input pattern to the wrong class. Even though the input pattern is closer to class mean of A, it is clearly in the cluster of class B.



3-2: One class having much larger standard deviation than the other class

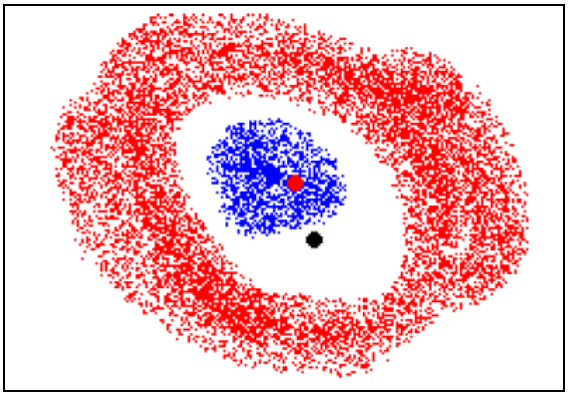
- ***Mahalanobis Classifier***

This is similar to the Euclidean classifier, however instead of using the Euclidean distance, we use the Mahalanobis distance which takes into account the statistical properties of the classes. Ray (2005d) defines it as:

$$\Delta_i^2 = (x - \mu_i)' \Sigma_i^{-1} (x - \mu_i)$$

Where μ_i is the mean feature vector of class i, x is the feature vector of input pattern x, and $\sum_i$ is the variance-covariance matrix of class i for the given feature vector. The input pattern x is then assigned to the class i if $\Delta_i^2 < \Delta_j^2$ for all classes j = 1...n.

This method is expected to perform better than Euclidean, since it takes into account the statistical properties of the classes. Unlike the Euclidean distance, this classifier will correctly assign the input pattern to class A in figure 3-2. However, there is another problem. This method will generally work quite well, as long as the data is from a Gaussian distribution. Consider figure 3-3, where there is class which makes a ring around another class. The variance-covariance matrix will not be effective in this case, because the point is clearly within the standard deviation of the outer-class. This classifier and the Euclidean distance classifier will not be able to efficiently classify input patterns with this graph, even though both classes are clearly separated.



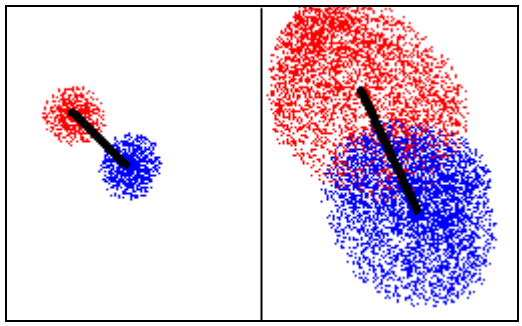
3-3: One class making ring around the other class

- ***K-Nearest Neighbors Classifier***

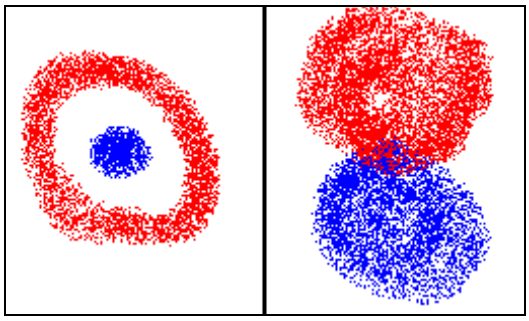
The K-Nearest neighbors classifier is quite different from the above two. Duda and Hart (1973) explain that this algorithm works by finding the k nearest neighbors of the input pattern (the neighbors consist of the sample data, whose classes are known) and assigning the input pattern to the class which occurs the most in those neighbors. For example, if the classes of the k = 3 nearest neighbors are (1,2,2) then the input pattern will be assigned class 2, since this occurs the most.

This method is based on the assumption that patterns of the same class will be close together. There is no specific k to choose to optimize the classifier; the optimal value will vary with the data-sets. However, as a general rule, increasing k too much will decrease the accuracy, since the classifier will begin relying more on the a'priori probabilities of the classes rather than the neighbors of the input pattern.

This is significantly more difficult to compute than the euclidean distance; however it should also be significantly better, since it takes into account the statistical properties of the classes. This method will choose feature subset A over B in figure 3-4. However, this method has similar drawbacks to the Mahalanobis classifier, consider figure 3-5. This will prefer feature subset B over A, even though subset A is better, since in A the distance between the class means is close to zero, which means Δ^2 is close to zero.



3-4: Classes with almost similar standard deviation



3-5: Classes with different standard deviations

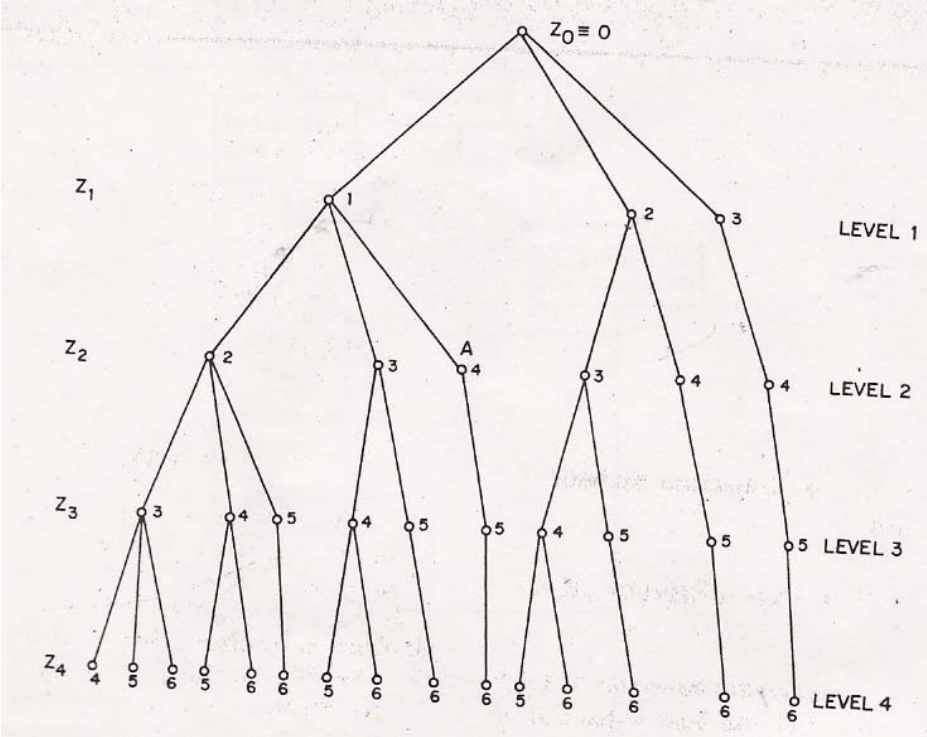
3.1.2.6. Feature Subset Searching

As explained earlier, it is not feasible to run the aforementioned criterion functions on every possible feature subset. Even though Cover and Campenhout (1977) explain that no non-exhaustive search can be guaranteed to select an optimal subset. We must search the possible subsets, with the possibility of discarding good or optimal subsets, to ensure that the algorithm completes in a time efficient manner. Some of the feature subset searching methods are:

- ***Branch and Bound***

The branch and bound algorithm is explained by Narendra and Fukunaga (1977) as an efficient feature subset search method which finds the optimal subset. Chen (2003); Narendra and Fukunaga (1977) and Clausen (1999) explain that the efficiency of this algorithm is due to the algorithm discarding many subsets which are guaranteed to be sub-optimal, without having to actually run the criterion function on the sets. The algorithm assumes monotonicity. For example, if we have two feature subsets A = {1,2,3} and B={2,3} where $B \subset A$, then $C(A) \geq C(B)$, that is, A is no worse than B, with respect to the criterion function being used.

The algorithm is explained by Narendra and Fukunaga (1977) as follows, from figure 3-6, the algorithm takes the right most branch, and descends down the single path until the bottom level (z4). At this point, the "bound" is set to C(1,2) (since we have discarded {3,4,5,6}) and the subset {1,2} is saved as the best subset seen so far. The algorithm then backtracks up the tree to the top (z0) and chooses the next branch, left of the one just processed, which is '2'. C({1,3,4,5,6}) is calculated, if it is smaller than the "bound" then, by monotonicity, we can discard that node and all the subsets below it, since these are guaranteed to be worse. We then backtrack up the tree and choose the next branch, which will be '1'.



3-6: Branch and Bound algorithm use

On the other hand, if C({1,3,4,5,6}) (node '2') it is greater than the bound, we can descend down the tree. If we reach the bottom, we update the "bound", save the subset as the best seen so far, and backtrack up the tree. The algorithm is finished when we have discarded and/or evaluated all the possible subsets, which is when we backtrack to the top from the left most branch.

Monotonicity is not a trivial issue, and it is clear that it plays an important role in this algorithm. Although the monotonicity criterion should hold for most data sets, it would be wrong to assume it holds for these all. Jain and Zongker (1997) point out that from the 'curse of dimensionality', in small sample size situations the monotonicity may not hold.

However, in the experiments by Hamamoto et al. (1990), the method is shown to work well even when the monotonicity criterion does not hold. The other problem with this method is that even though it is substantially more efficient than exhaustive search, Jain and Zongker (1997) point out that its worst case complexity is exponential, which becomes an issue when dealing with large feature sets.

Branch and Bound is a commonly used and well known method to solve combinatorial problems, such as feature subset selection.

- ***Sequential Forward Selection***

Forward selection works by selecting the best single feature and then progressively adding features which enhance a particular criterion the most. The algorithm is explained by Pudil et al. (1994) as:

1. We begin with the current subset Current = Ø; and Available = $\{x_1, x_2, ...x_m\}$, where x_i is feature i and m is the total number of features.

2. Find x_i where J(Current $\cup$ x_i) is the largest for all x_j in Available

3. Current = Current $\cup$ x_i

4. Available = Available − x_i

5. If the subset size is smaller than our requirement, Go to step 2, else terminate the algorithm

While this method is fast, it discards many subsets at an early stage. Thus, it is unlikely to provide an optimal subset. As Guyon and Elisseeff (2003) state that a feature might seem useless by itself but can be quite useful with another particular feature. This is the method that is used in this research and implementation.

- ***Sequential Backward Selection***

This algorithm effectively works in reverse of Forward Selection. We start with the complete set of features, and progressively discard the least useful feature until we have a subset of the required size.

This is more difficult to implement computationally, and also takes longer to run than forward selection.

- ***Sequential Floating Selection***

This method essentially puts together Forward Selection and Backward Selection into the same algorithm. The problem with both methods, explains Pudil et al. (1994), is that once a decision has been made to add (or remove) a feature, there is no chance to undo the step

later down the track. The sequential floating selection solves this by running a number of backward steps after each forward step, as long as each backward step results in a better subset. The algorithm given by Pudil et al. (1994) essentially works as follows:

Let $X_k = \{x_1, x_2, ...x_k\}$, the current subset of size k. We begin with $X_0 = Ø$.

1. (a) Find x where $\max(J(X_k \cup x))$

(b) $X_{k+1} = X_k \cup x$

(c) $k = k + 1$

Apply one forward step, that is, select the feature which when added to the current subset, the resulting subset enhances the criterion the most.

2. (a) Find x where $\max(J(X_k - x))$

(b) If $J(X_k - x) > J(X_{k-1})$ then

i. $X_{k-1} = X_k - x$

ii. $k = k - 1$

iii. Goto step 2

(c) else Goto step 1

Apply one backward step, that is, select the feature which when removed from the current subset, the resulting subset enhances the criterion the most. If that subset of size $k - 1$ performs better than the previous subset of size $k - 1$, remove the feature from the subset and re-apply step 2. Otherwise, goto step 1.

The algorithm terminates once we have a subset of the required size. Although, this algorithm is not guaranteed to produce an optimal subset, unlike Branch and Bound, yet this does not assume monotonicity. Pudil et al. (1994) states that this algorithm produces results very close to the Branch and Bound algorithm, but requires less computational effort.

- ***Exhaustive Search***

Exhaustive search is mostly used on small-data sets. A benefit of exhaustive search is that it allows us to find out which subset the criterion function deems to be optimal, which

can be a useful thing to compare. It can also be used to compare the results of different feature subset searching methods.

3.2. Previous Work

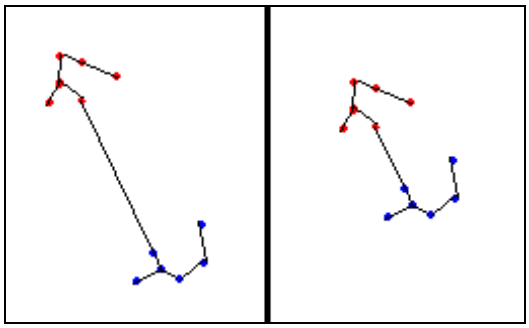
An algorithm for clustering by Smith and Jain (1984) plots random points 'y' within the bounds of the sample data 'x', then proceeds to build a minimum spanning tree across this data. The algorithm then counts the number of edges that joined a random point to a sample point. Smaller the number, the better. The idea behind it is, according to Jain and Dubes (1988), that well clustered data should have a high number of x-x and y-y joins compared to random data. Although this can be used to measure how clustered the data is, it cannot be used to determine whether the clusters are from the same class or not, which is critical to selecting feature subsets.

A method proposed by Friedman and Rafsky (1979) uses minimum spanning trees to evaluate whether two sets of n-dimensional data are from the same distribution. A minimum spanning tree is built across the data points, and edges which connect data from one distribution to the other are removed. If many edges are removed, then the data from the distributions are mixed up together, and so these must come from the same distribution.

This approach was used for feature subset selection in [1]. They try to find a feature subset which best shows that the sets of data come from different classes. Given a feature subset to evaluate, a minimum spanning tree is built across the sample data. The edges leading from one class to another are removed, and tallied. The more edges that are removed, the worse the feature subset is. This approach also handles the distributions shown in figures 3-4 and 3-5.

However, it does not take into account the weight of the edges, so that if one feature subset has two clusters connected by an edge length 2*x and another subset has 2 clusters connected by an edge length of x, both will yield the same value when the subset with the edge of 2*x should be selected. In addition, the algorithm does not take into account the distances between samples of the same class, which should be minimized. Dense clusters

of the same class are preferred to sparse clusters. Figure 3-7 shows an example of this situation. Although it can be argued that either subset can be selected, and a classifier should provide excellent results in both cases, it is important to remember that the data being used is sample data, and that live data could have a larger spread, and noisier input patterns.



3-7: Two class distributions in which the first distribution should be preferred

A new method of feature subset selection was proposed by Don and Kothari (2003). They introduced the concept that an n-dimensional classification problem can be visualized in (n+1) dimensions with the class label as the extra dimension. An example provided was a 2D graph of a 2 feature problem, made into a 3D graph, with the class label as the third dimension. They explain that the smoothness of the class label surface can be quantified, and therefore the classifiability can be measured (smoother the surface the better). This method is good for rating subsets when samples from different classes are close together and it will be able to classify the distributions shown in figures 3-4 and 3-5.

However, similar to the method proposed by Friedman and Rafsky (1979), if two of subsets exist where clusters of samples from different classes are not touching, the class label surface will be smooth for both subsets. This is not desirable if one of the subsets keeps the classes separated further away than the other.

3.3. Approach Implemented

Minimum spanning trees, as explained by Friedman and Rafsky (1979), are well known for providing superior descriptions of sets of points in pattern recognition.

A minimum-spanning tree is a sub-graph of a weighted, connected and undirected graph. It is acyclic, connects all the nodes in the graph, and the sum of all of the weight of all of its edges is minimum, that is, there is no other spanning tree, or sub-graph which connects all the nodes and has a smaller sum. If the weights of all the edges are unique, the minimum spanning tree is unique.

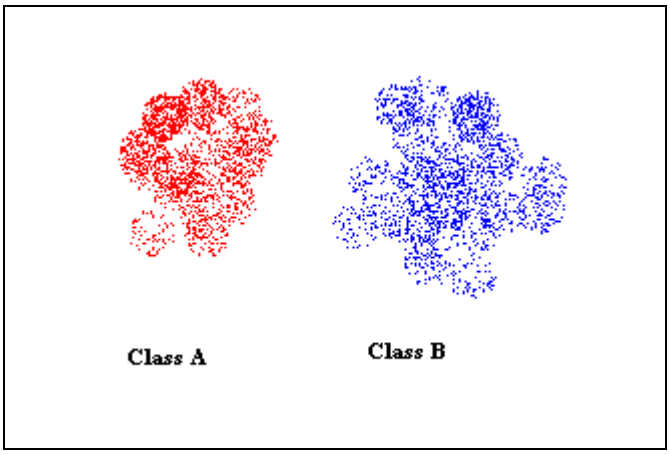
When constructing a tree in reference to the samples used for pattern recognition, the nodes in the tree will represent the samples, and the axis of the n-dimensional graph represents the n features.

The Euclidean minimum spanning tree is the minimum spanning tree where the weight of an edge connecting two nodes is the Euclidean distance between the nodes. The distance between sample pairs will therefore be the euclidean distance between their feature vectors.

A minimum spanning tree can be constructed by first making a graph such that every node is connected with every other node. This can be done by calculating the distance between each pair of nodes and thus making an edge between these with the weight equal to the distance between the nodes. This process of making a graph takes $O(N^2)$ time.

The rationale for using Minimum Spanning Trees for selecting a feature subset is that we want to find such clusters of samples in which samples belonging to one class are close together. These close clusters should then be at a greater distance from all other clusters. As shown in the following image, we want such clusters which have samples close together so that new samples can be recognized and classified based on the nearest neighbors method.

Our main purpose is to reduce the distance between samples belonging to one class. Since the main objective of minimum spanning trees is to eliminate all high distance edges, minimum spanning trees can be used for finding and evaluating such subsets and clusters.



3-8: Two dense classes well separated

After the graphs are made by selecting different features, we have to make MST's on those graphs which will initially eliminate all the larger edges. Then, we have to select such an MST from all the available ones in which the distances between samples belonging to one class are very small whereas the distances belonging to samples of different classes are large. In other words, we have to select such an MST that has close clusters of one class separated by large distances from other clusters. This will ensure good accuracy during recognition and classification, since all dense clusters will be separated by relatively large distances. This will be the basis of the criterion function which will be used to evaluate the different MST's and consequently choosing one MST.

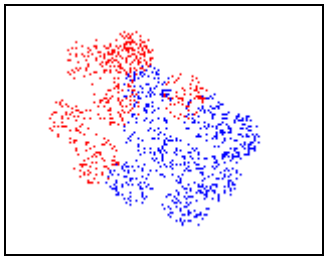
Another important issue is the feature subset searching. The process of choosing features to be evaluated and added in the feature subset is called feature subset searching. There are many ways of choosing features which are described as under:

3.3.1. Proposed Criterion Function

The research mainly focuses on finding a criterion function on the basis of which one feature subset is said to be 'better' than another feature subset.

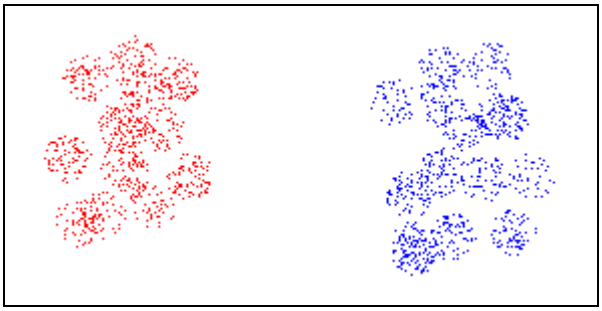
A graph is built first of all based on S, where S is the set of all training samples. The complete feature set is represented by F. The set of features selected so far is represented by SF. In the beginning, SF = Ø.

One by one, a feature is added to the set SF. On the basis of the selected features' values, the euclidean distance between each sample pair is calculated and an edge made between these samples (node). The weight of this newly created edge is the calculated euclidean distance. Subsequently, a minimum spanning tree is constructed on this graph using Prim's algorithm. Based on the criterion function C(x), the cost of the minimum spanning tree is calculated. The minimum spanning tree with the smallest cost is then chosen and this whole process of adding features to the SF set continues until the number of selected features is equal to the number of features that were specified to be used.



3-9: Classes not separated distinctively from each other

The criterion function C(x) takes into account the distance between samples belonging to one class as well as the distance between samples belonging to different classes. It calculates the edge distance percentages of samples of one class and then calculates the fraction of these edges with respect to the percentage of edge distances of samples belonging to different classes. This fraction basically shows if the class clusters are close together or not. If this fraction is large, it means that distance between samples of one class was larger than the distances between samples of different classes as depicted in figure 3-9.



3-10: Dense class clusters separated distinctively

If this fraction is small, it means that the clusters of samples belonging to one class are far away from each other, as depicted in figure 3-10.

Thus, the smaller this fraction or the criterion function value, the 'better' this MST will be and will result in greater accuracy of recognition and classification. This criterion function will also be able to handle and correctly classify distributions such as shown in figure 3-7.

The criterion function is further described in detail in section 3.4.

3.3.2. Data Structures of MST

The data structures used for representing graphs and other entities are described as under:

- A two dimensional integer array to store the training samples (*SampleArray*).
    - The number of rows represents the number of samples.
    - The number columns is the number of features that describe each sample.
    - The values in the array represent the actual value of the feature for that particular sample.
- A two dimensional integer array to store the output class values for the training samples (*SampleOutput*).
    - The number of rows represents the number of samples.
    - The number columns is the number of output for each sample; in this case the number of columns is 1.
    - The values in the array represent the actual class output of that particular sample.
- Similar two arrays for the testing samples and testing output (*TestArray* and *TestOutput*).
- A two dimensional distance matrix for storing the graph (*Graph*).
    - The number of rows is the total number of samples

  - The number of columns is the total number of samples
  - The values in the matrix represent the euclidean distance between the pair of samples. This matrix will be filled completely since the distance between all sample pairs will be calculated and stored.

- A two dimensional distance matrix for storing the minimum spanning tree (*MST*).
  - The number of rows is the total number of samples
  - The number of columns is the total number of samples
  - The values in the matrix represent the euclidean distance between the pair of samples. This matrix will be partially filled because when the MST will be built; many edges would have been removed resulting in a zero value for all the sample pairs that do not have an edge between them.
- A one dimensional incidence matrix for storing the selected features (*SelectedFeatures*).
  - The number of rows represents the number of features.
  - The values of the array represent whether a particular feature has been selected yet or not.
- Two dimensional matrix for storing the nearest nodes information for recognition and classification (*NearestNode*)
- Two dimensional matrix for storing the nearest node output values for classification (*NearestOutput*)
- Integer variables for storing the following entities:
  - *Samples* - Total number of nodes (samples)
  - *TotalFeatures* – The total number of features that describe each sample
  - *TestSamples* – The number of samples of testing data
- Double type variables for storing the following:
  - *TP* – True Positive Rate

- *FP* – False Positive Rate

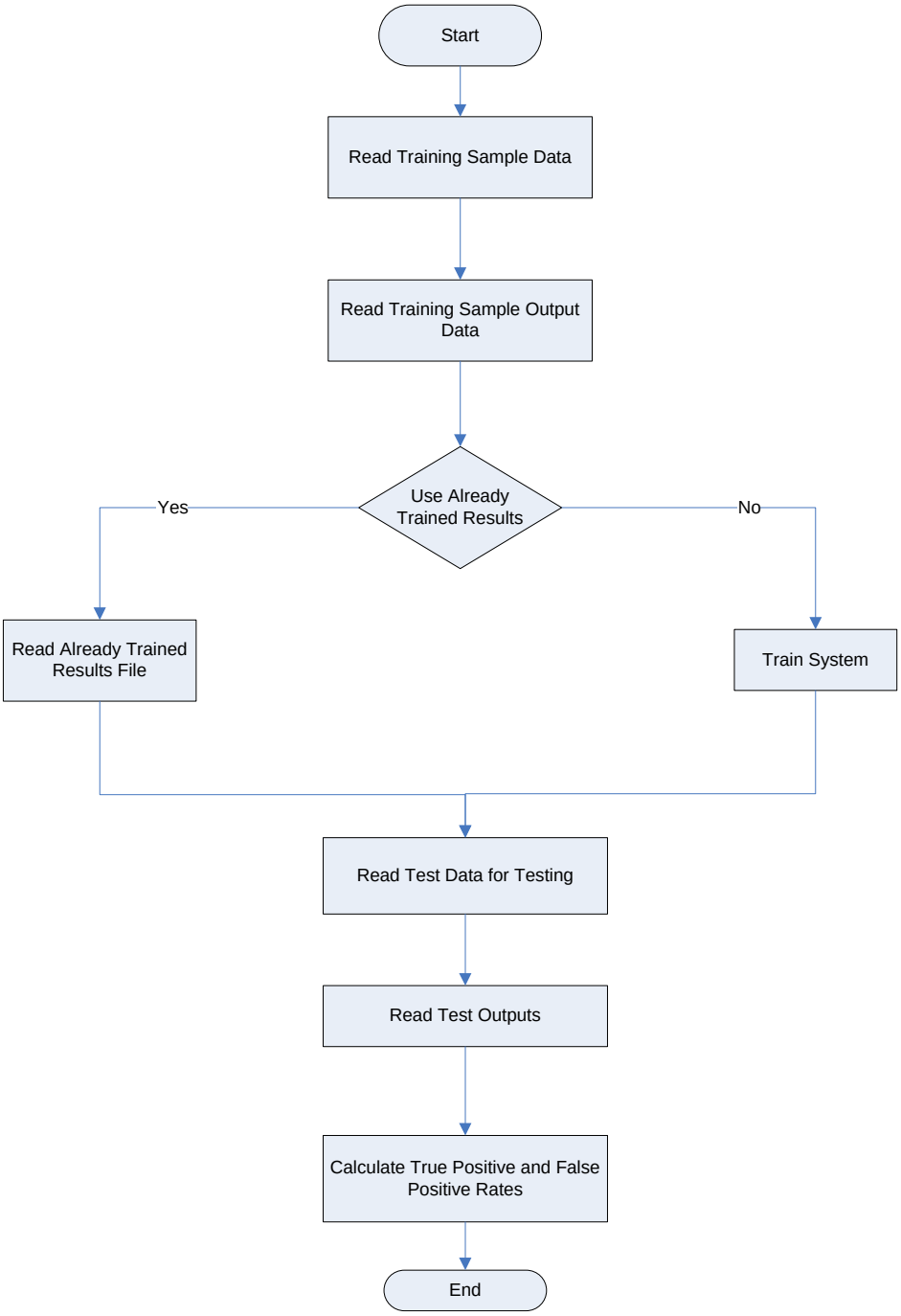
3.4. Implementation Details

The system has been implemented in Visual C++ language.

The overall flow of the system is described as under:

- The training sample data file is loaded into the system.
- The training file is read and all the information is stored in the relevant data structures.
- The training sample data output is loaded into the system.
- The output values of sample data are stored in the relevant data structures.
- If the system has already been trained for a particular data set and the user wishes to use that trained information instead of training the system again, the trained information is loaded in the system.
- If the system is to be trained again, the user specifies the maximum number of features that are to be used to train the system.
- The system is then trained by selecting the best possible feature subset.
- The user can then also write this trained information in a file for later use.
- The system is then tested for some testing data.
- The testing data file and output file is loaded into the system.
- The user specifies the 'k' value that will be used in the k-nearest neighbors algorithm for classifying the testing samples.
- After classifying all the test data samples, the true positive and false positive rates are calculated to analyze the efficiency of the algorithm

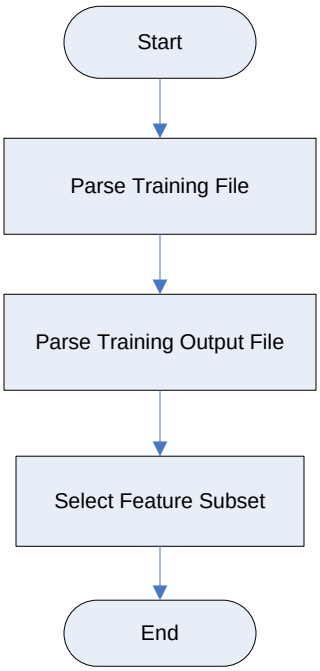
Overall Flow of the System



The process of training the system can be described as under:

- First the training data file is parsed using a parser.
- Each line of the data file represents one sample.
- The parser reads each line of data from the file, stores the separate feature values of the sample in the relevant data structure until the complete file is read.
- The training output file is fed into the parser.
- Each line of this file represents the output class value for each sample.
- The parser reads output of each sample and stores it in the relevant data structure until the file is completely read.
- The process of selecting the feature subset is called and consequently the best feature subset is selected.

Flow of Training System



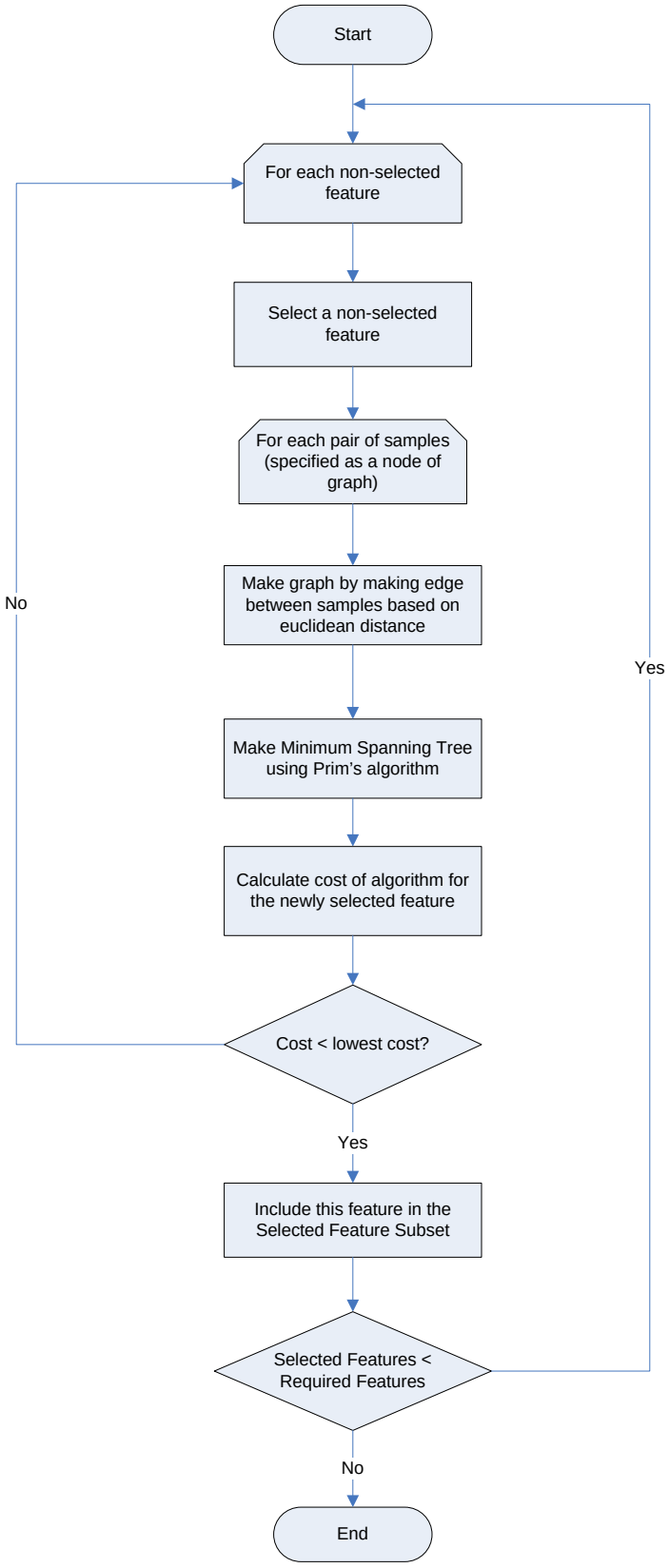
The process of Feature Subset Selection can be described as under:

- The selected feature set *SF* initially is empty.
- For each non-selected feature, a feature is chosen and added in the array *CheckFeatures.*
- Each sample represents a node of the graph.
- Based on the values of *CheckFeatures*, a graph is built among all the samples.
- The euclidean distance from each sample to every other sample is calculated, based on the feature values, and consequently an edge is made between those samples (nodes).
- All this information is stored in the relevant graph data structure.
- After the graph has been built completely, the minimum spanning tree is built on the graph based on Prim's algorithm.
- The efficiency of the MST is calculated so that the best MST is chosen.
- The criterion function C(x) for evaluating an MST is described as under:
    - The weights of edges between nodes of one class are added together. Let it be represented by IntraEdgeWeight_1.
    - The cardinality of the edges between nodes belonging to one class is represented by IntraEdges_1.
    - Similarly, the weights of edges between nodes of the second class are added together. Let it be represented by IntraEdgeWeight_2.
    - The cardinality of the edges between nodes belonging to one class is represented by IntraEdges_2.
    - The edges between nodes belonging to different classes are added and represented by InterEgdesWeight.
    - The cardinality of the edges between nodes belonging to different classes is represented by InterEdges.

- The following costs are calculated:
  - $CostPerClass_1 = IntraEdgesWeight_1 / IntraEdges_1$
  - $CostPerClass_2 = IntraEdgesWeight_2 / IntraEdges_2$
  - $CostPerInterClasses = InterEdgesWeight / InterEdges$
  - $C(x) = (CostPerClass_1 + CostPerClass_2) / (CostPerClass_1 + CostPerClass_2 + CostPerInterClasses)$

- The value of this C(x) is stored along with the feature which was being checked for addition in the feature subset.
- This feature is then removed from the *CheckFeatures* array.
- Similarly, another feature is added to *CheckFeatures*, and the process of making graph and MST is repeated.
- The criterion function is calculated and its subsequent value stored with the feature being checked.
- This process continues until all the unchecked features have been taken into consideration and their criterion function values calculated.
- The criterion function values for all the features are then compared and the feature for which the criterion function value was the lowest is chosen and added in the final selected feature subset *SF*.
- The whole process of checking unchecked features continues until the desired number of features has been added in the selected feature subset SF.

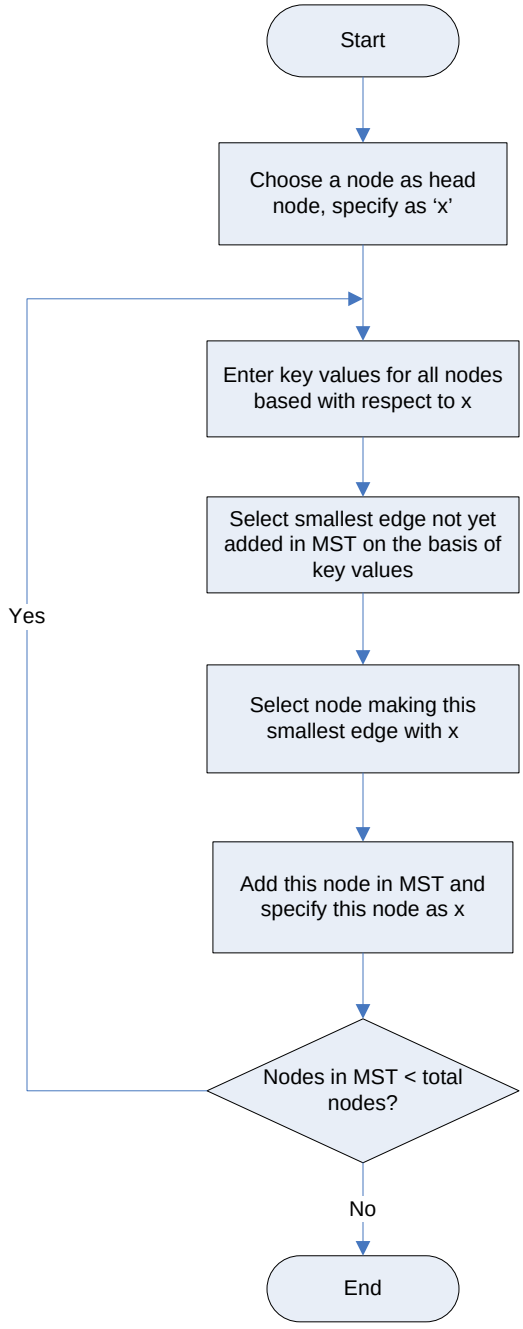
Flow of Feature Subset Selection



The process for making the minimum spanning tree (MST) is described as under:

- The graph of all the samples (nodes) connected with each other through the euclidean distance, based on the feature values, is taken to convert into a minimum spanning tree.
- A node in the graph is chosen at random as the root of the MST. Let it be represented by ‘x’.
- Initially the MST is empty. After addition of ‘x’, it has just one node.
- The smallest distance from ‘x’ to all its adjacent nodes.is calculated These distances are represented by *key* values.
- The smallest key from these key values is selected. This edge will be added in the MST as well as the node that was connecting ‘x’ through this edge.
- This newly added node in the MST is specified as ‘x’.
- The process is then again repeated by calculating key values with respect to the new ‘x’ and consequently choosing the smallest edge and adding it to MST.
- This process is repeated until all the nodes of the graph are added in the MST.

Flow for Making Minimum Spanning Tree (MST) using Prim's Algorithm



The system is then tested by calculating the true positive (TP) and false positive (FP) rates by using test data. The process of testing the trained system is described as under:

- First the testing data file is parsed using a parser.
- Each line of the data file represents one sample.
- The parser reads each line of data from the file and stores the separate feature values of the sample in the relevant data structure until the complete file is read.
- The testing output file is fed into the parser.
- Each line of this file represents the output class value for each sample.
- The parser reads output of each sample and stores it in the relevant data structure until the file is completely read.
- For each test sample, the euclidean distance between the test sample feature vector and all the training samples are calculated one by one, taking into consideration only those features which were chosen in the feature subset SF.
- The user also specifies an odd value for ‘k’. This ‘k’ value will be used in predicting the output value of the sample using the k-nearest neighbors method.
- After the euclidean distance of the test sample with all training samples has been calculated, the ‘k’ lowest distances and their corresponding training samples will be chosen.
- The output values of these ‘k’ training samples are considered and the majority value of the output is chosen as the predicted class value of the test sample.
- Four variables A, B, C and D will be used for calculating true positive and false positive rates.
- If the predicted class value is of class 1 then,
    - If the actual class output for that test sample (as read from the test output file) is of class 1, variable D is incremented. This will represent the test samples which actually belonged to class 1 and were also predicted correctly as belonging to class 1.

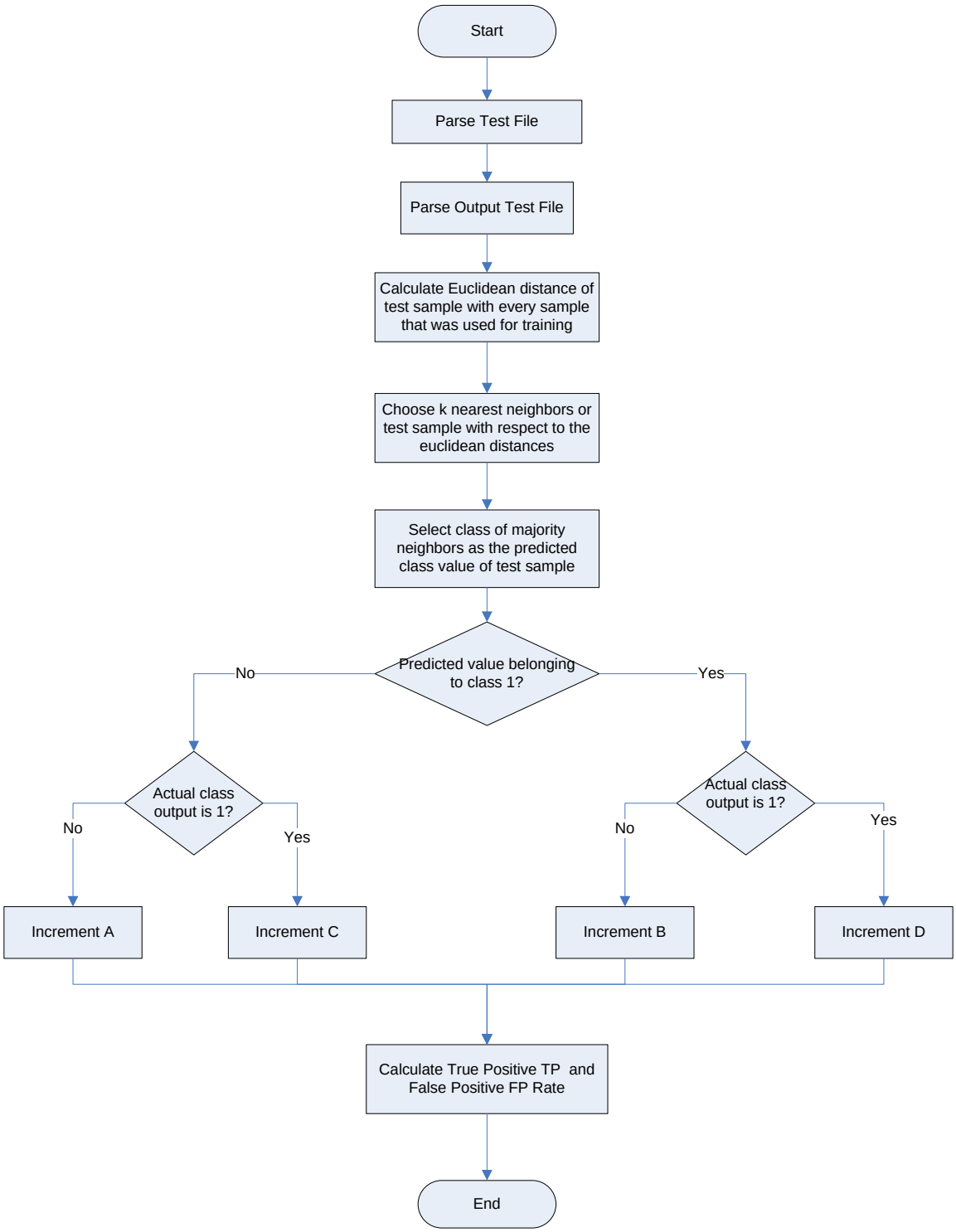
- Else, if the actual class output of that test sample is of class 2, variable B is incremented. This will represent those test samples which actually belonged to class 2 but were incorrectly predicted as belonging to class 1.

- Else, if the predicted class output is of class 2 then,
  - If the actual class output for that test sample (as read from the test output file) is of class 1, variable C is incremented. This will represent the test samples which actually belonged to class 1 but were incorrectly predicted as belonging to class 2.
  - Else, if the actual class output of that test sample is of class 2, variable A is incremented. This will represent those test samples which actually belonged to class 2 and were predicted correctly as belonging to class 2.
- This process is repeated for each test sample and the variables A, B, C and D updated after each test sample is taken into consideration.
- The true positive rate TP is then calculated as:

  TP = D / (C+D)
- The false positive rate FP is then calculated as:

  FP = B / (A+B)
- The higher the True Positive rate (TP) and the lower the False Positive rate (FP), the greater the accuracy with which the system will be able to recognize and classify samples correctly.

Flow for Calculating True Positive (TP) and False Postive (FP) Rates



Chapter 4 : Results and Discussion

4.1. Data Sets

Three different data sets were used for training and testing of the system. All these datasets are two-class sets i.e. the samples belong to two classes. The details of each data set are as under:

4.1.1. Hepatitis Data Set

- Title: Hepatitis Domain
- Sources:
    - Donor: G.Gong (Carnegie-Mellon University) via Bojan Cestnik, Jozef Stefan Institute
    - Date: November, 1988
- Past Usage:
    - Diaconis,P. & Efron,B. (1983). Computer-Intensive Methods in Statistics. Scientific American, Volume 248.
    - Gail Gong reported a 80% classification accuracy
    - Cestnik,G., Konenenko,I, & Bratko,I. (1987). Assistant-86: A Knowledge-Elicitation Tool for Sophisticated Users. In I.Bratko & N.Lavrac (Eds.) Progress in Machine Learning, 31-45, Sigma Press.
    - Assistant-86: 83% accuracy
- Number of Instances: 155
- Number of Attributes: 20 (including the class attribute)
- Attribute information:
    - Class: DIE, LIVE
    - AGE: 10, 20, 30, 40, 50, 60, 70, 80

- SEX: male, female
- STEROID: no, yes
- ANTIVIRALS: no, yes
- FATIGUE: no, yes
- MALAISE: no, yes
- ANOREXIA: no, yes
- LIVER BIG: no, yes
- LIVER FIRM: no, yes
- SPLEEN PALPABLE: no, yes
- SPIDERS: no, yes
- ASCITES: no, yes
- VARICES: no, yes
- BILIRUBIN: 0.39, 0.80, 1.20, 2.00, 3.00, 4.00
- ALK PHOSPHATE: 33, 80, 120, 160, 200, 250
- SGOT: 13, 100, 200, 300, 400, 500,
- ALBUMIN: 2.1, 3.0, 3.8, 4.5, 5.0, 6.0
- PROTIME: 10, 20, 30, 40, 50, 60, 70, 80, 90
- HISTOLOGY: no, yes

- Missing Attribute Values:

  Attribute Number: Number of Missing Values:

  4: 1

  6: 1

  7: 1

  8: 1

9:	10
10:	11
11:	5
12:	5
13:	5
14:	5
15:	6
16:	29
17:	4
18:	16
19:	67

- Class Distribution:
  - DIE: 32
  - LIVE: 123

4.1.2. Wisconsin Diagnostic Breast Cancer (WDBC) Data Set

- Title: Wisconsin Diagnostic Breast Cancer (WDBC)
- Source Information
  - Creators: Dr. William H. Wolberg, General Surgery Dept., University of Wisconsin, Clinical Sciences Center, Madison, WI 53792 wolberg@eagle.surgery.wisc.edu
  - W. Nick Street, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706 street@cs.wisc.edu 608-262-6619
  - Olvi L. Mangasarian, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706 olvi@cs.wisc.edu

- Donor: Nick Street
- Date: November 1995

- Past Usage:
  - First usage: W.N. Street, W.H. Wolberg and O.L. Mangasarian Nuclear feature extraction for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905, pages 861-870, San Jose, CA, 1993.
  - Literature: O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and prognosis via linear programming. Operations Research, 43(4), pages 570-577, July-August 1995.
  - Medical literature: W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techniques to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77 (1994) 163-171.
  - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Image analysis and machine learning applied to breast cancer diagnosis and prognosis. Analytical and Quantitative Cytology and Histology, Vol. 17 No. 2, pages 77-87, April 1995.
  - W.H. Wolberg, W.N. Street, D.M. Heisey, and O.L. Mangasarian. Computerized breast cancer diagnosis and prognosis from fine needle aspirates. Archives of Surgery 1995;130:511-516.
  - W.H. Wolberg, W.N. Street, D.M. Heisey, and O.L. Mangasarian. Computer-derived nuclear features distinguish malignant from benign breast cytology. Human Pathology, 26:792--796, 1995.
  - http://www.cs.wisc.edu/~olvi/uwmp/mpml.html
  - http://www.cs.wisc.edu/~olvi/uwmp/cancer.html

- Relevant information

- Features were computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. These describe characteristics of the cell nuclei present in the image.
- A few of the images can be found at http://www.cs.wisc.edu/~street/images/
- Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes.
- The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].
- This database is also available through the UW CS ftp server: ftp ftp.cs.wisc.educd math-prog/cpo-dataset/machine-learn/WDBC/

- Number of instances: 569
- Number of attributes: 32 (ID, diagnosis, 30 real-valued input features)
- Attribute information
  - ID number
  - Diagnosis (M = malignant, B = benign)
  - Ten real-valued features are computed for each cell nucleus:
    - radius (mean of distances from center to points on the perimeter)
    - texture (standard deviation of grey-scale values)

- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2 / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

- Several of the papers listed above contain detailed descriptions of how these features are computed. The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, and field 23 is Worst Radius.

- Missing attribute values: none
- Class distribution: 357 benign, 212 malignant

4.1.3. Wisconsin Prognostic Breast Cancer (WPBC) Data Set

- Title: Wisconsin Prognostic Breast Cancer (WPBC)
- Source Information
  - Creators: Dr. William H. Wolberg, General Surgery Dept., University of Wisconsin, Clinical Sciences Center, Madison, WI 53792 wolberg@eagle.surgery.wisc.edu
  - W. Nick Street, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706, street@cs.wisc.edu 608-262-6619
  - Olvi L. Mangasarian, Computer Sciences Dept., University of Wisconsin, 1210 West Dayton St., Madison, WI 53706, olvi@cs.wisc.edu

- Donor: Nick Street
- Date: December 1995

- Past Usage:

  Various versions of this data have been used in the following publications:

  - W. N. Street, O. L. Mangasarian, and W.H. Wolberg. An inductive learning approach to prognostic prediction. In A. Prieditis and S. Russell, editors, Proceedings of the Twelfth International Conference on Machine Learning, pages 522--530, San Francisco, 1995. Morgan Kaufmann.
  - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and prognosis via linear programming. Operations Research, 43(4), pages 570-577, July-August 1995.
  - W.H. Wolberg, W.N. Street, D.M. Heisey, and O.L. Mangasarian. Computerized breast cancer diagnosis and prognosis from fine needle aspirates. Archives of Surgery 1995;130:511-516.
  - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Image analysis and machine learning applied to breast cancer diagnosis and prognosis. Analytical and Quantitative Cytology and Histology, Vol. 17 No. 2, pages 77-87, April 1995.
  - W.H. Wolberg, W.N. Street, D.M. Heisey, and O.L. Mangasarian. Computer-derived nuclear "grade" and breast cancer prognosis. Analytical and Quantitative Cytology and Histology, Vol. 17, pages 257-264, 1995.
  - http://www.cs.wisc.edu/~olvi/uwmp/mpml.html
  - http://www.cs.wisc.edu/~olvi/uwmp/cancer.html

- Relevant information

  - Each record represents follow-up data for one breast cancer case. These are consecutive patients seen by Dr. Wolberg since 1984, and include only those cases exhibiting invasive breast cancer with no evidence of distant metastases at the time of diagnosis.

- The first 30 features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. These describe characteristics of the cell nuclei present in the image.
- A few of the images can be found at http://www.cs.wisc.edu/~street/images/
- The separation described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes.
- The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].
- The Recurrence Surface Approximation (RSA) method is a linear programming model which predicts Time To Recur using both recurrent and non-recurrent cases.
- This database is also available through the UW CS ftp server: ftp ftp.cs.wisc.educd math-prog/cpo-dataset/machine-learn/WPBC/

- Number of instances: 198
- Number of attributes: 34 (ID, outcome, 32 real-valued input features)
- Attribute information
  - ID number
  - Outcome (R = recur, N = non-recur)
  - Time (recurrence time if field 2 = R, disease-free time if field 2 = N)

- Ten real-valued features are computed for each cell nucleus:
  - radius (mean of distances from center to points on the perimeter)
  - texture (standard deviation of gray-scale values)
  - perimeter
  - area
  - smoothness (local variation in radius lengths)
  - compactness (perimeter^2 / area - 1.0)
  - concavity (severity of concave portions of the contour)
  - concave points (number of concave portions of the contour)
  - symmetry
  - fractal dimension ("coastline approximation" - 1)
- The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 4 is Mean Radius, field 14 is Radius SE, field 24 is Worst Radius.
- Values for features 4-33 are recoded with four significant digits.
- Tumor size - diameter of the excised tumor in centimeters
- Lymph node status - number of positive axillary lymph nodes observed at the time of surgery

- Missing attribute values:
  - Lymph node status is missing in 4 cases.
- Class distribution: 151 non-recur, 47 recur

4.2. Results

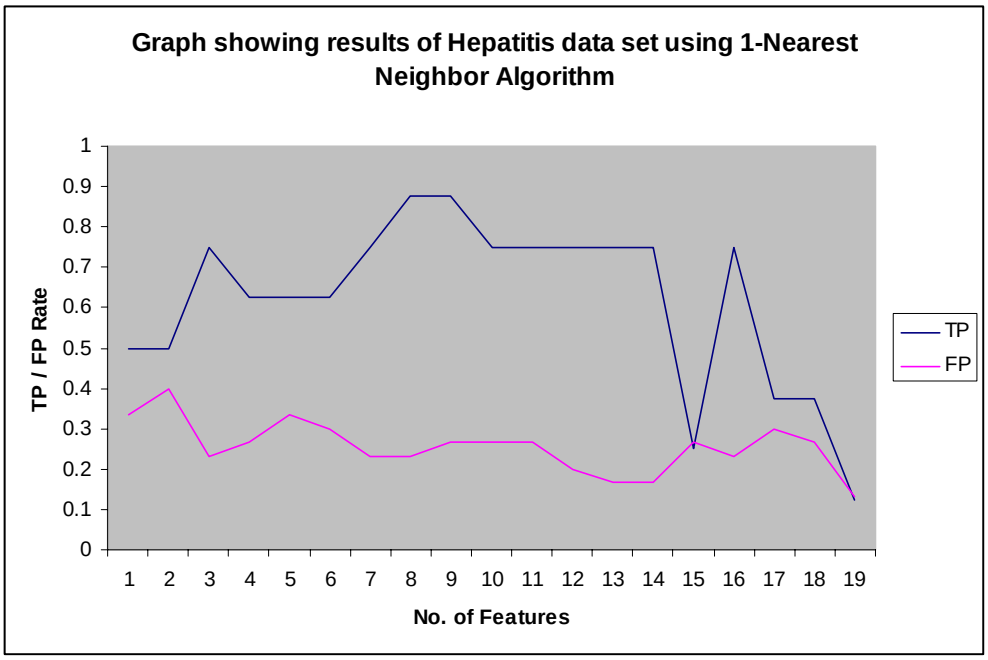
4.2.1. Hepatitis

The Hepatitis data set was used to train the system. Minimum spanning tree approach was used to select the feature subset. The program was run various times by choosing different values for the number of features to be selected. The time it takes for the system to train depends on the number of features that are to be selected and the number of training samples. If this number is small, it takes less time as compared to the situation in which more features are to be selected. The time taken ranges from 40 milliseconds (when 1 feature is to be selected) to 1 second and 37 milliseconds (when the maximum number of features i.e. 18 is selected). The recognition and classification was then done using 1-nearest, 3-nearest and 5-nearest neighbors methods. The following results for the true positive and false positive rates were obtained by selecting different number of features:

	1-nearest		**3-nearest**		**5-nearest**	
Features	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
1	0.5	0.333	0.5	0.266	0.375	0.2
2	0.5	0.4	0.5	0.3	0.375	0.03
3	0.75	0.233	0.5	0.166	0.375	0.03
4	0.625	0.266	0.625	0.233	0.375	0.066
5	0.625	0.333	0.625	0.333	0.375	0.1
6	0.625	0.3	0.625	0.2	0.375	0.1
7	0.75	0.233	0.75	0.2	0.375	0.1
8	**0.875**	**0.233**	0.75	0.166	0.375	0.066
9	0.875	0.266	0.5	0.166	0.375	0.066
10	0.75	0.266	0.5	0.166	0.375	0.03
11	0.75	0.266	0.5	0.166	0.375	0.03
12	0.75	0.2	0.5	0.133	0.25	0.033
13	0.75	0.166	0.5	0.1	0.25	0.033
14	0.75	0.166	0.375	0.1	0.25	0.033
15	0.25	0.266	0.375	0.133	0.125	0.033
16	0.75	0.233	0.625	0.066	0.5	0.033
17	0.375	0.3	0.375	0.033	0.25	0
18	0.375	0.266	0.375	0.033	0.25	0
19	0.125	0.133	0.375	0.066	0.25	0

4-1: Results showing TP and FP Rates of Hepatitis Data Set

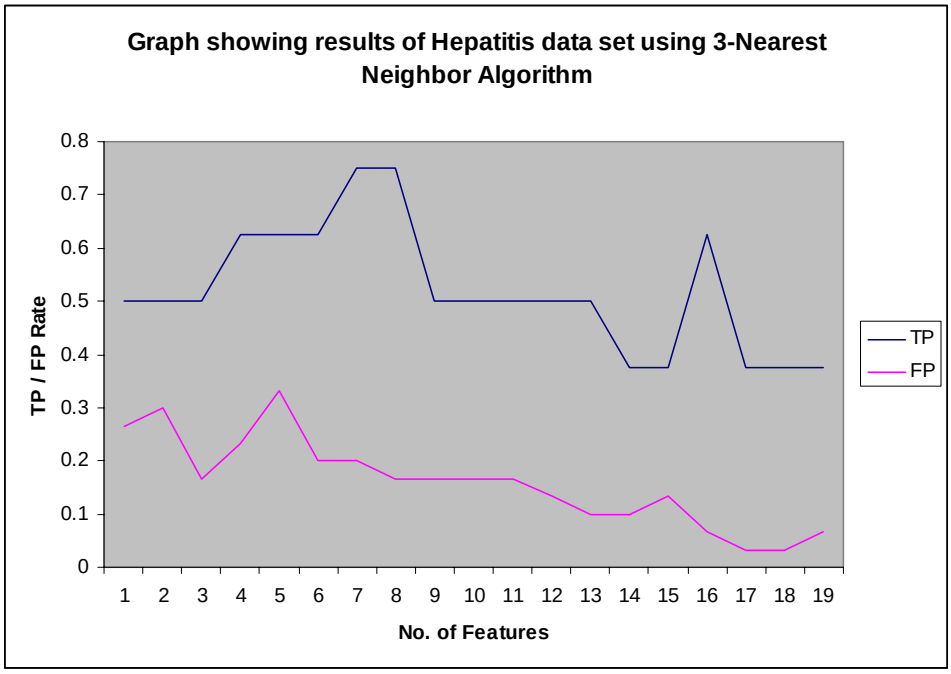
The following graph shows the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 1-nearest neighbors method was used.



4-2: TP and FP Rate using 1-Nearest Neighbors Method

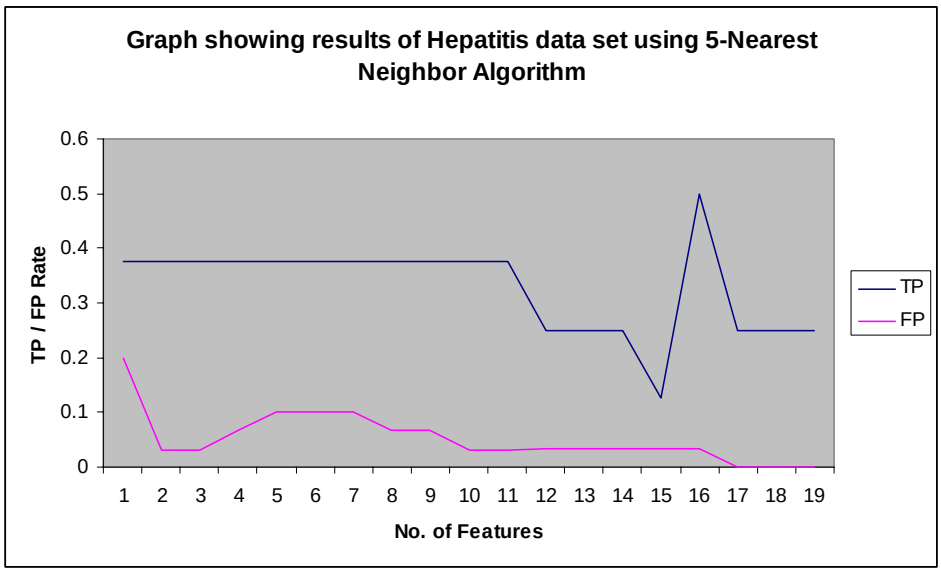
As shown in the graph, the best results are being shown when 8 features are selected with a TP rate of 0.875 and FP rate of 0.233. The system works very well in classifying objects of this data set with an accuracy rate of almost 90% by utilizing only 40% of the features.

The following graphs show the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 3-nearest neighbors and 5-nearest neighbors methods were used.



4-3: TP and FP Rate using 3-Nearest Neighbors Method

In case of 3-nearest neighbors classification, the best performance is seen when 8 features are selected with TP rate of 0.75 and FP rate of 0.166.



4-4: TP and FP Rate using 5-Nearest Neighbors Method

In case of 5-nearest neighbors method, although the false positive rate is very less which is desirable; the true positive rate is also not very good with the best value of TP of 0.5 when 16 relevant features are selected. So, in this data set, 1-nearest neighbor gives the best results.

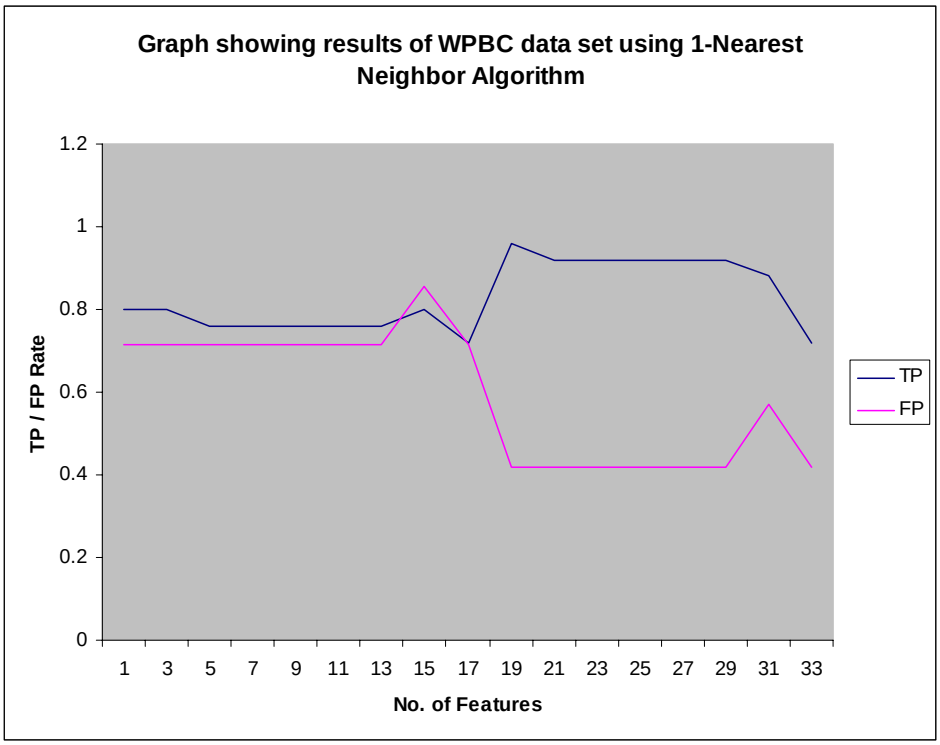
4.2.2. WPBC

The WPBC data set was then used to train the system. The program was run various times by choosing different values for the number of features to be selected. When one feature was to be selected, the training time was 1 second whereas when the maximum number of features i.e. 33 was chosen. The training time was 7 seconds 42 milliseconds. The recognition and classification was done using 1-nearest, 3-nearest and 5-nearest neighbors methods. The following results for the true positive and false positive rates were obtained by selecting different number of features:

	1-nearest		3-nearest		5-nearest	
Features	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
1	0.8	0.714	0.92	0.714	0.96	0.714
3	0.8	0.714	0.92	0.714	0.96	0.714
5	0.76	0.714	0.88	0.714	0.96	0.714
7	0.76	0.714	0.88	0.714	0.96	0.714
9	0.76	0.714	0.88	0.714	0.96	0.714
11	0.76	0.714	0.88	0.714	0.96	0.714
13	0.76	0.714	0.88	0.714	0.84	0.714
15	0.8	0.857	0.84	0.714	0.92	0.857
17	0.72	0.714	0.84	0.714	0.92	0.857
19	0.96	0.42	0.88	0.42	0.76	0.42
21	0.92	0.42	0.88	0.42	0.8	0.42
23	0.92	0.42	0.88	0.42	0.8	0.42
25	0.92	0.42	0.88	0.42	0.8	0.42
27	0.92	0.42	0.88	0.42	0.8	0.42
29	0.92	0.42	0.88	0.42	0.8	0.42
31	0.88	0.57	0.84	0.42	0.88	0.42
33	0.72	0.42	0.64	0.57	0.52	0.714

4-5: Results showing TP and FP Rates of WPBC Data Set

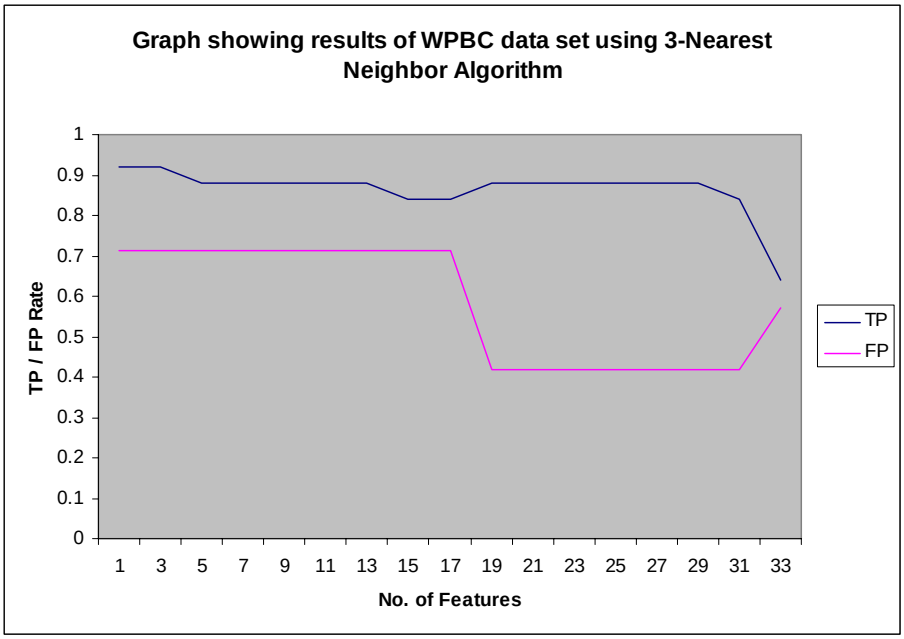
The following graph shows the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 1-nearest neighbors method was used.



4-6: TP and FP Rate using 1-Nearest Neighbors Method

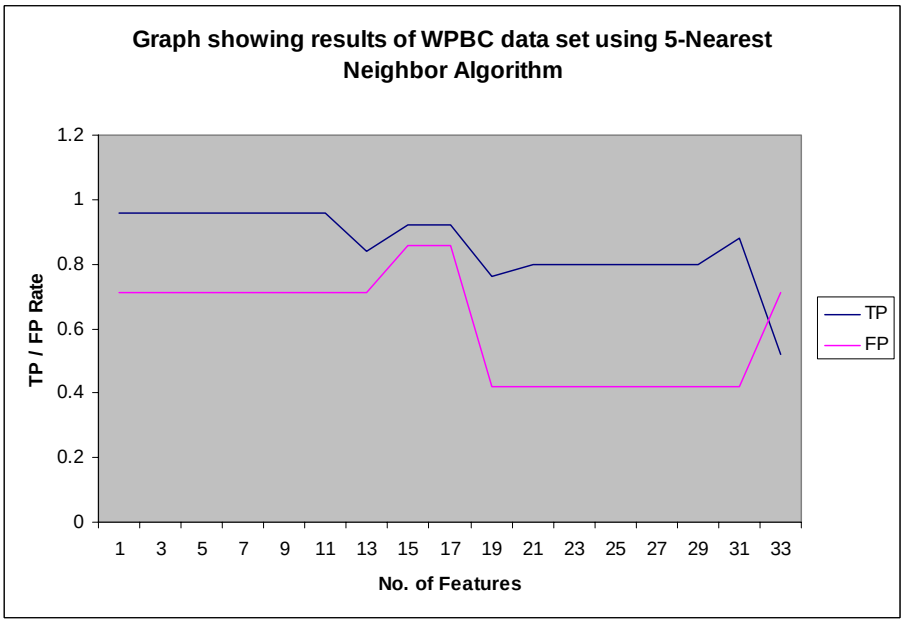
As shown in the graph, the best results are being shown when 19 features are selected with a TP rate of 0.96 and FP rate of 0.42. The system works very well in classifying objects of this data set with an accuracy rate of almost 96% by utilizing 57% of the features.

The following graphs show the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 3-nearest neighbors and 5-nearest neighbors methods were used.



4-7: TP and FP Rate using 3-Nearest Neighbors Method

In case of 3-nearest neighbors classification, the best performance is seen with minimum 19 features are selected with TP rate of 0.88 and FP rate of 0.42.



4-8: TP and FP Rate using 5-Nearest Neighbors Method

In case of 5-nearest neighbors method, the best TP rate of 0.96 is achieved with only 1 feature selected but at the same time, the FP rate is also very high, i.e. 0.714. Overall, the program does not work very well with this 5-nearest neighbors method.

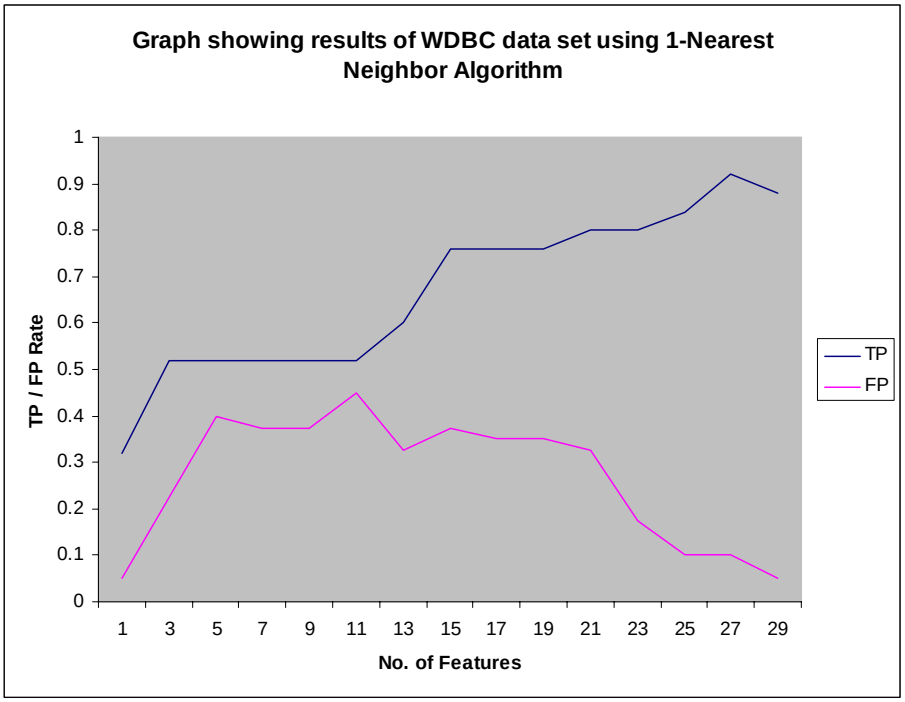
4.2.3. WDBC

The WDBC data set was used to train the system. The program was run various times by choosing different values for the number of features to be selected. When one feature was to be selected, the time taken was 3 seconds and 40 milliseconds whereas when the maximum number of features is selected, the training time becomes 1 minute, 9 seconds and 2 milliseconds. The recognition and classification was then done using 1-nearest, 3-nearest and 5-nearest neighbors methods. The following results for the true positive and false positive rates were obtained by selecting different number of features:

	1-nearest		3-nearest		5-nearest	
Features	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
1	0.32	0.05	0.4	0.075	0.44	0.175
3	0.52	0.225	0.6	0.15	0.64	0.275
5	0.52	0.4	0.6	0.25	0.6	0.275
7	0.52	0.375	0.6	0.225	0.6	0.275
9	0.52	0.375	0.64	0.275	0.64	0.2
11	0.52	0.45	0.6	0.325	0.68	0.275
13	0.6	0.325	0.64	0.25	0.6	0.1
15	0.76	0.375	0.56	0.45	0.64	0.475
17	0.76	0.35	0.6	0.45	0.64	0.475
19	0.76	0.35	0.6	0.4	0.6	0.475
21	0.8	0.325	0.64	0.4	0.64	0.45
23	0.8	0.175	0.72	0.15	0.72	0.2
25	0.84	0.1	0.8	0.075	0.8	0.1
27	0.92	0.1	0.84	0.075	0.8	0.05
29	0.88	0.05	0.84	0.05	0.8	0

4-9: Results showing TP and FP Rates of WDBC Data Set

The following graph shows the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 1-nearest neighbors method was used.

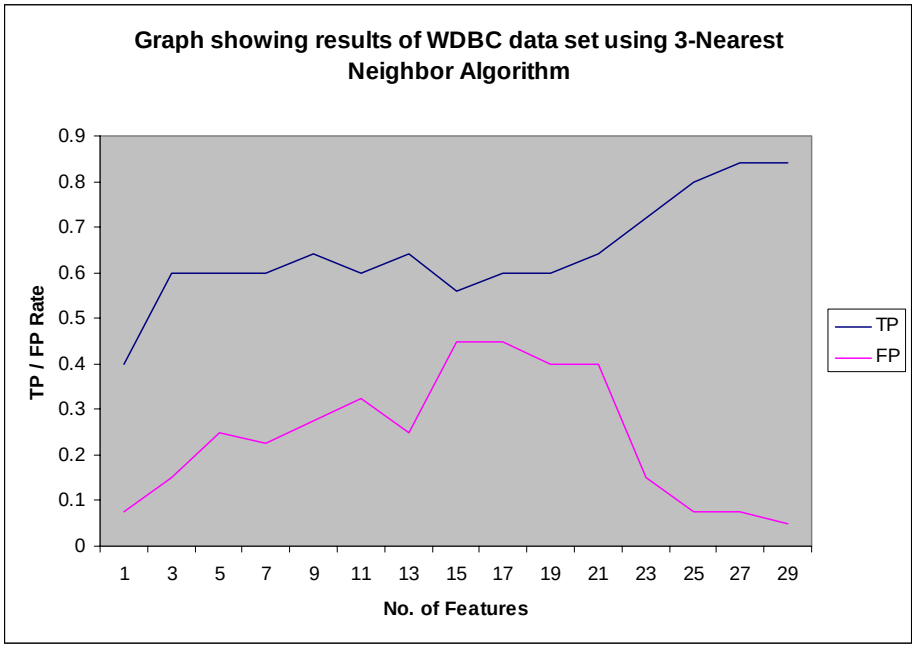


4-10: TP and FP Rate using 1-Nearest Neighbors Method

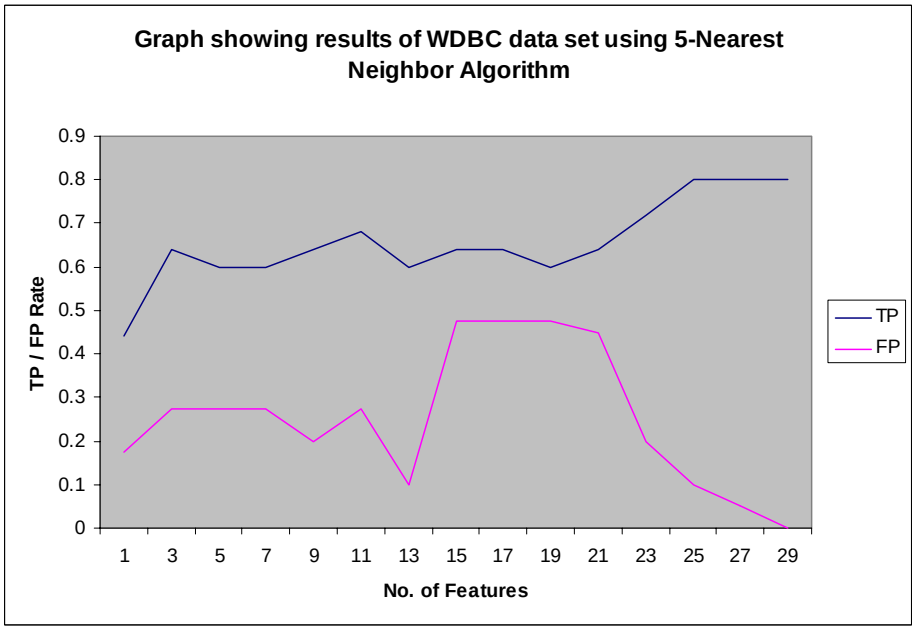
As shown in the graph, the best results are being shown when 27 features are selected with a TP rate of 0.8 and FP rate of just 0.05. The system works very well in classifying objects of this data set with an accuracy rate of almost 80% but has the drawback that it has selected a lot of features which would not result in time efficiency. A relatively less accurate result could be achieved when 15 features are selected with TP rate of 0.76 and FP rate of 0.375.

The following graphs show the true positive and false positive rates when a certain number of features are selected for training. For recognition and classification, 3-nearest neighbors and 5-nearest neighbors methods were used. Similar to 1-nearest neighbor method, it gives very good TP and FP rates but at the expense of choosing and selecting

most of the features of the samples. This in turn, does not result in good efficiency as time is not saved when most of the original features are used.



4-11: TP and FP Rate using 3-Nearest Neighbors Method



4-12: TP and FP Rate using 5-Nearest Neighbors Method

4.3. Comparison with other Researcher's Work using Different Classifiers

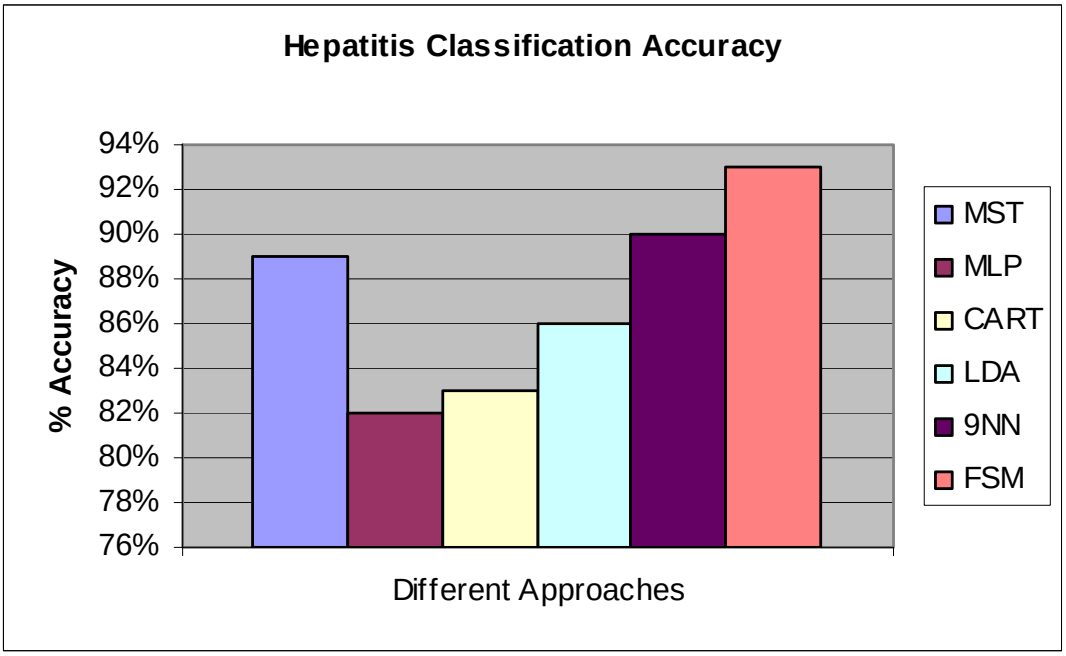
The maximum classification accuracy rates of MST approach and other classifiers used by different researchers were compared. The results were then tabulated and plotted in bar graphs.

4.3.1. Hepatitis

The maximum accuracy rate achieved using MST approach is plotted against the different classifiers used by other researchers. In comparison to the other classifiers, MST approach came in 3rd place with maximum accuracy of 89% whereas FSM resulted in an accuracy rate of 93%.

	MST	MLP	CART	LDA	9NN	FSM
Accuracy	89%	82%	83%	86%	90%	93%

4-13: Percentage Accuracy of MST and Others for Hepatitis Dataset



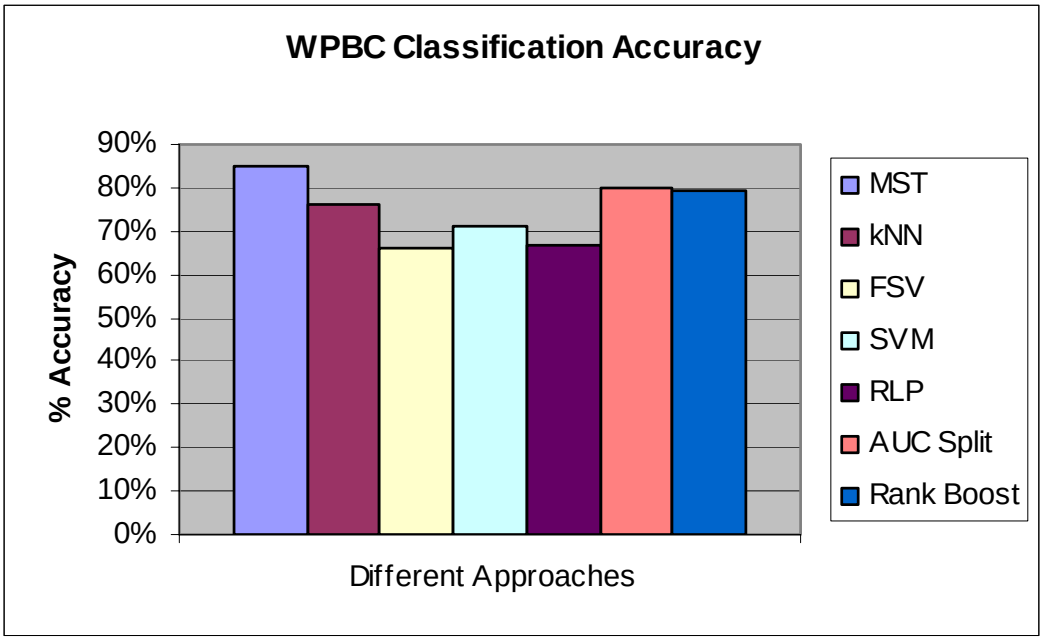
4-14: Comparison of MST approach with others for Hepatitis Dataset

4.3.2. WPBC

The maximum accuracy rate achieved by MST approach for the WPBC dataset turned out to be 85% which is more than any other approach used by other researchers. This was plotted in the form of bar graphs from where it could be seen that MST worked best for WPBC dataset. The second best accuracy rate of 80% was recorded by AUC Split approach.

	MST	kNN	FSV	SVM	RLP	AUC Split	Rank Boost
Accuracy	85%	76%	66%	71%	67%	80%	79%

4-15: Percentage Accuracy of MST and Others for WPBC Dataset



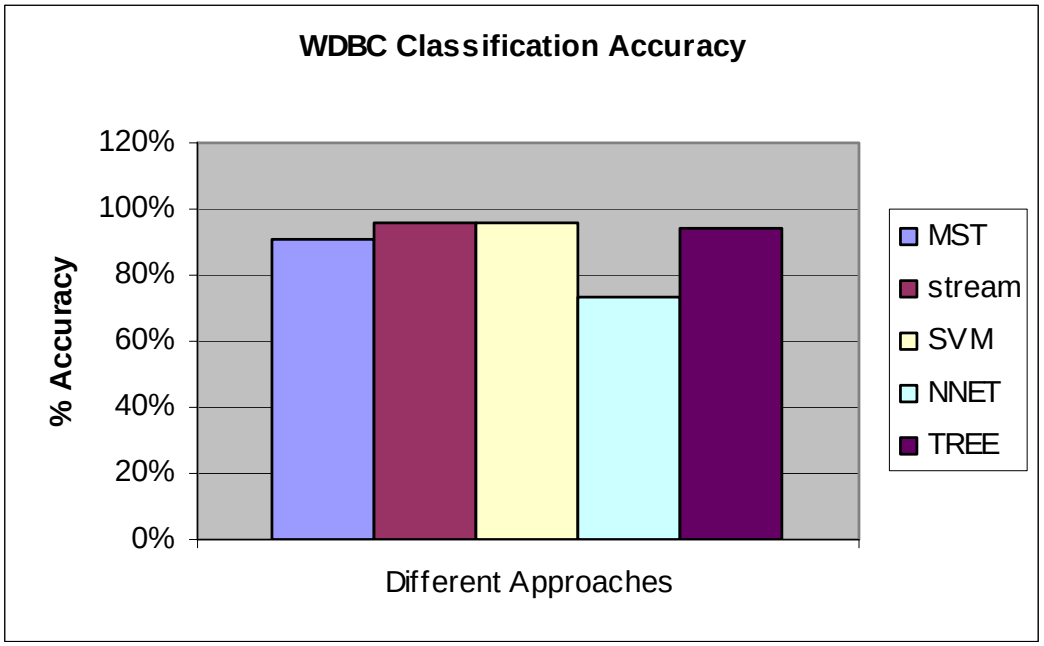
4-16: Comparison of MST approach with others for WPBC Dataset

4.3.3. WDBC

The maximum accuracy rate achieved by MST approach for WDBC dataset was 91%. Although this accuracy rate was higher than the rate achieved for other datasets; however, even higher accuracy rates of up to 96% have been recorded for this dataset by other different approaches. This has been outlined in the following table and the bar graph.

	MST	Stream	SVM	NNET	TREE
Accuracy	91%	96%	96%	73%	94%

4-17: Percentage Accuracy of MST and Others for WDBC Dataset



4-18: Comparison of MST approach with others for WDBC Dataset

Chapter 5 : Conclusion and Future Work

This thesis presents the implementation of a disease recognition system using minimum spanning trees for feature subset selection in pattern recognition and classification. The system has been tested on three data sets which provide good results in recognition and classification.

5.1. Graphs and Trees

Graphs are mathematical structures used to model pair-wise relations between objects from a certain collection. A graph refers to a collection of vertices or 'nodes' and a collection of *edges* that connect pairs of vertices. Graphs have been used widely in different areas of computer science including the field of Artificial Intelligence. Graphs provide a very detailed and meaningful representation which can further be converted into different types of trees and other forms. This, in turn, helps in changing the domain of the problem thus providing better options for its solution. In pattern recognition, graphs have been used for modeling relations between different samples that are used for training and recognition. A **tree** is a graph in which any two vertices are connected by *exactly one* path. Alternatively, any connected graph with no cycles is a tree. Trees are widely used in computer science data structures such as binary search trees, heaps, tries, Huffman trees for data compression, etc.

5.2. Minimum Spanning Trees

A spanning tree of a graph is a sub-graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. Some algorithms exist for converting a graph into a minimum spanning tree e.g. Kruskal's algorithm, Prim's algorithm, Boruvka's algorithm etc.

The use of MST in pattern recognition is rather a new field and some research is being done in this regard. MST's are used in this implementation for feature subset selection during the training phase of patter recognition.

5.3. MSTs for Feature Subset Selection

Pattern recognition involves three steps namely, pattern representation, feature selection and classification. Pattern representation is the process of gathering the data that will be used in training the system. Feature selection is the process of choosing a subset of the entire feature set. This subset will be the one that is 'best' in terms of classifying the objects. The last stage is that of classification in which the objects are classified based on the feature set and the knowledge it has gathered. K-nearest neighbors method is used for classification purposes.

To choose the best feature subset, MST's were used. The rationale for using MST's for this purpose was that MST's are constructed by deleting all the longer edges from the graph. The approach used in this thesis is that we want to have such clusters of sample data in which samples belonging to one class are close together (smaller edges). These clusters of different classes should then be separated from clusters of other classes through relatively larger distances. In other words, the intra-class distances should be small as compared to the inter-class distances. The resulting MST's are then evaluated through a criterion function. The criterion function calculates the density and distance of edges belonging to one class and then calculates its fraction with respect to the inter class distance. The smaller the fraction, the better is the MST according to the criterion function being used.

5.4. Future Work

The use of MST in feature subset selection is a rather new field and requires more research and work. The technique used in this thesis provides good results up to some extent but more work can be done in this regard to increase the accuracy and efficiency of the system. Some of the issues that need to be addressed are as under:

- Prim's algorithm has been used to make MST's for a particular feature set. The process of building MST's takes a lot of time, so a parallel implementation of Prim's algorithm will result in better time efficiency.
- The criterion function that is used to evaluate MST's can be further improved so as to get better accuracy rates of almost 100%.
- Since this approach was tested for supervised learning, an extension would be to use it for unsupervised learning in which the class outputs of the training samples are not known beforehand.
- Another extension would be to use data sets of samples belonging to more than 2 classes.

References

[1] Mike Farah, *Feature Subset Selection using Minimum Cost Spanning Trees*, 2004.

[2] Surendra K. Singhi, Huan Liu, *Feature Subset Selection Bias for Classification Learning*, Proceedings of the 23rd International Conference of Machine learning, 2006

[3] George Forman, *An extensive Empirical Study of Feature Selection Metrics for Text Classification,* Journal of Machine Learning Research 3, 2003

[4] YongSeog Kim, W. Nick Street, Filippo Menczer, *Evolutionary Model Selection in Unsupervised Learning*, Intelligent Data Analysis IOS Press, 2002

[5] Julia Handl, Joshua Knowles, *Feature Subset Selection in Unsupervised Learning via Multiobjective Optimization,* International Journal of Computational Intelligence Research, 2006

[6] Isabelle Guyon, Jiwen Li, Theodor Mader, Patrick A. Pletscher, George Schneider, Markus Uhr, *Feature Selection with the CLOP package*, 2006.

[7] Zehang Sun, George Babis, Ronald Miller, *Object Detection using Feature Subset Selection*, Journal of Pattern Recognition Society, 2004.

[8] Isabelle Guyon, Andre Elisseeff, *An Introduction to Variable and Feature Selection*, Journal of Machine Learning Research 3, 2003

[9] Sudhir Varma, Richard Simon, *Interative Class Discovery and Feature Selection using Minimal Spanning Trees*, BMC Bioinformatics, 2004.

[10] Nicolaj Sondberg-Madsen, Casper Thomsen, Jose M. Pena, *Unsupervised Feature Subset Selection*, 2004.

[11] Ekaterina Gonina, Laxmikant V. Kal´, *Parallel Prim's algorithm on dense graphs with a novel extension*, 2007

[12] Jeffrey L. Solka, Avory C. Bryant and Edward J. Wegman, *Text Data Mining with Minimal Spanning Trees*, 2005

[13] Zeev (Vladimir) Volkovich, Zeev Barzily, Ba_sak Akteke- Ozturk , Gerhard-Wilhelm Weber, *Cluster Stability Using Minimal Spanning Trees*, 2005.

[14] Alan Gibbons, *Algorithmic Graph Theory* (1985), Cambridge University Press.

[15] Chartrand, Gary, *Introductory Graph Theory*, Dover.

[16] Biggs, N.; Lloyd, E. & Wilson, R. Graph Theory, 1736-1936 Oxford University Press, 1986

[17] Diestel, Reinhard (2005), *Graph Theory* (3rd ed.), Berlin, New York: Springer-Verlag

[18] Otter, Richard (1948), "The Number of Trees", *Annals of Mathematics, Second Series* 49

[19] Roman Dementiev, *"Engineering an External Memory Minimum Spanning Tree Algorithm"*

[20] Eppstein, David (1999). "Spanning trees and spanners". *Handbook of Computational Geometry*: 425–461, Elsevier.

[21] Wu, Bang Ye; Chao, Kun-Mao (2004). *Spanning Trees and Optimization Problems*. CRC Press

[22] Garey, Michael R.; Johnson, David S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman

[23] Devijver, P. A. and Kittler, J. (1982). Pattern Recognition: A Statistical Approach, Prentice Hall

[24] Don, M. and Kothari, R. (2003). Feature subset selection using a new definition of classifiability, Pattern Recognition Letters 24

[25] Duda, R. O. and Hart, P. E. (1973). Pattern Classification and Scene Analysis, Wiley-Interscience

[26] Friedman, J. H. and Rafsky, L. C. (1979). Multivariate generalizations of the waldwolfowitz and smirnov two-sample tests, The Annals of Statistics 7: 697–717.

[27] Ray, S. (2005a, 2005b, 2005c, 2005d). Bayesian classification. Lecture notes from CSE456, “Pattern recognition”, School of Computer Science and Software Engineering, Monash University

[28] Tou, J. T. and Gonzalez, R. C. (1974). Pattern Recognition Principals, Addison-Wesley.

[29] Friedman, M. and Kandel, A. (1999). Introduction to Pattern Recognition,World Scientific

[30] Cover, T. M. and Campenhout, J. M. V. (1977). On the possible ordering in the measurement selection problem, IEEE Transactions on Systems Man and Cybernetics 9: 657–661.

[31] Hamamoto, Y., Uchimura, S., Matsuura, Y., Kanaoka, T. and Tomita, S. (1990). Evaluation of the branch and bound algorithm for feature selection, Pattern Recognition Letters 11: 453–456.

[32] Pudil, P., Novovicova, J. and Kittler, J. (1994). Floating search methods for feature selection with nonmonotonic criterion functions, Pattern Recognition Letters 15: 1119–1125.

[33] Smith, S. P. and Jain, A. K. (1984). Testing for uniformity in multidimensional data, IEEE Transactions on Pattern Analysis and Machine Intelligence 6: 73–81.

[34] Jain, A. K. and Dubes, R. C. (1988). Algorithms for Clustering Data, Prentice Hall.

[35] Jain, A. K. and Zongker, D. (1997). Feature selection: Evaluation, application, and small sample performance, IEEE Transactions on Pattern Analysis and Machine Learning 19: 153–158.

[36] Narendra, P. M. and Fukunaga, K. (1977). A branch and bound algorithm for feature subset selection, IEEE Transactions on Computers 26: 917–922.