

CHOOSING THE BEST HEURISTIC FOR A NP-PROBLEM

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

**Master of Engineering
in
Computer Science and Engineering**



Thapar University, Patiala.

By
Narendhar Maaroju
(80732015)

Under the supervision of

Dr. Deepak Garg
Asst. Professor
CSED

JUNE 2009
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY,
PATIALA – 147004.

CERTIFICATE

I hereby certify that the work which is being presented in the thesis report entitled, "**Choosing the B Heuristic for a NP-Problem**", submitted by me in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Deepak Garg** and refers other researcher's works which are duly listed in the reference section.

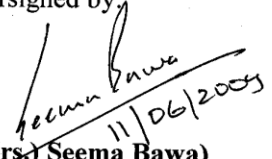
The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

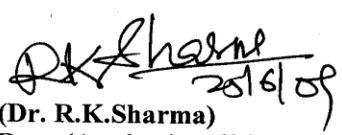

(Narendhar Maaroju)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.


(Dr. Deepak Garg)
Asst. Professor
Computer Science and Engineering Department
Thapar University, Patiala.

Countersigned by:


(Dr. (Mrs.) Seema Bawa)
Professor & Head
Computer Science & Engineering Department
Thapar University,
Patiala.


(Dr. R.K. Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

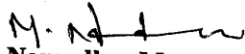
ACKNOWLEDGEMENT

No volume of words is enough to express my gratitude towards my guide **Dr. Deepak Garg**, Asst. Professor, Department of Computer Science & Engineering, Thapar University, Patiala, who has been very concerned and has aided for all the materials essential for the preparation of this thesis report. He has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to **Dr. (Mrs.) Seema Bawa**, Head of Department, Computer Science & Engineering Department and **Mrs. Inderveer Channa**, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis work.

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.


Narendhar Maaroju
(80732015)

ABSTRACT

Nowadays computers are used to solve incredibly complex problems. But in order to manage a problem we should develop an algorithm. Sometimes the human brain is not able to accomplish this task. Moreover, exact algorithms might need centuries to solve a formidable problem. In such cases heuristic algorithms that find approximate solutions but have acceptable time and space complexity play indispensable role. In present, all known algorithms for NP-complete problems are requiring time that is exponential in the problem size. Heuristics are a way to improve time for determining an exact or approximate solution for NP-problems. In our paper we want to analyze what are the possible heuristics available for NP-problems and we explain the characteristics and performance of each heuristic. Finally we analyze efficient heuristic out of all available heuristics for different NP-problems. One objective is that, after applying different heuristics for a particular NP-problem, a set of guidelines can be given that how a particular category of heuristics is better for a particular set of problems.

TABLE OF CONTENTS

CERTIFICATE.....	i
ACKNOWLEDGEMENT.....	ii
ABSTRACT.....	iii
TABLE OF CONTENTS.....	iv
LIST OF FIGURES.....	vi
LIST OF TABLES.....	vii
1. INTRODUCTION.....	1
1.1 Introduction.....	1
1.2 Organization of Thesis.....	4
2. ALGORITHMS & COMPLEXITY.....	5
2.1 Introduction to Algorithms.....	5
2.2 Complexity classes.....	5
2.3 Examples of different NP-Problems.....	8
2.3.1 Boolean satisfiability problem (SAT).....	8
2.3.2 Fifteen Puzzle.....	9
2.3.3 The Temporal Knapsack Problem.....	10
2.3.4 Minesweeper.....	11
2.3.5 Tetris.....	11
2.3.6 Hamiltonian cycle problem.....	13
2.3.7 Traveling salesman problem(TSP).....	13
2.3.8 Subgraph isomorphism problem.....	14
2.3.9 Subset sum problem.....	15
2.3.10 Clique Problem.....	16
2.3.11 Vertex Cover Problem.....	16
2.3.12 Independent Set problem.....	18
2.4 Hierarchy of complexity classes.....	19
2.5 NP-Problems in different fields.....	19
3. HEURISTIC ALGORITHMS.....	20
3.1 Definition of Heuristic.....	20
3.2 Different heuristic strategies.....	22
3.2.1 Exhaustive search.....	23

3.2.2 Local search.....	24
3.2.3 Divide and Conquer.....	24
3.2.4 Branch-and-Bound.....	25
3.2.5 Dynamic Programming.....	26
3.2.6 Greedy Technique.....	27
3.3 Heuristic techniques for NP-problems.....	28
3.3.1 Hill-Climbing.....	28
3.3.2 Simulated Annealing.....	29
3.3.3 Tabu Search.....	30
3.3.4 Swarm Intelligence.....	32
3.3.5 Evolutionary Algorithms.....	33
3.3.6 Neural Networks.....	35
3.3.7 Support Vector Machines.....	36
4. PROBLEM STATEMENT.....	39
5. RESULTS & DISCUSSION.....	40
6. CONCLUSION.....	50
ANNEXURES	
I. References.....	52
II. List of Publications.....	54

LIST OF FIGURES

Figure 2.1: Shows the starting position for the 15-puzzle.....	10
Figure 2.2: Minesweeper game beginning and at the end of a completed game.....	12
Figure 2.3: The seven basic Tetris pieces.....	12
Figure 2.4: Tetris on the Nintendo Game Boy system.....	13
Figure 2.5: The mapping of Hamilton Cycle Problem Graph.....	14
Figure 2.6: Shows Isometric Trees.....	16
Figure 2.7: Shows Isometric Graphs.....	17
Figure 2.8: Shows a Clique Problem of size 3.....	17
Figure 2.9: Shows different vertex vectors.....	18
Figure 2.10: Shows the relation between different NP-problems.....	19
Figure 2.11: The known relationships between the complexity classes.....	20
Figure 3.1: Shows order of evaluation of nodes in DFS.....	24
Figure 3.2: Shows order of evolution of nodes in BFS.....	24
Figure 3.3: Finding the shortest path in a graph using optimal substructure.....	27
Figure 3.4: The process of Evolution Algorithms.....	33
Figure 3.5: The Basic Neural Unit.....	36
Figure 3.6: The Basic Components of Neural Networks	36
Figure 3.7: Two-Dimensional SVM Example.....	37
Figure 3.8: Classification of Support Vectors.....	38

LIST OF TABLES

Table 3.1 Illustrative Tabu Search Applications.....	31
Table 5.1: Comparison of different heuristic strategies for TSP.....	41
Table 5.2: Comparison of different heuristic strategies for Time Table Problem.....	43
Table 5.3: Comparison of different heuristic strategies for GEP.....	45
Table 5.4: Comparison of different heuristic strategies for Vertex Cover Problem.....	46
Table 5.5: Comparison of different heuristic strategies for Mean Flow Time Open Shop Scheduling.....	48

CHAPTER 1

INTRODUCTION

1.1 Introduction

The most important among a variety of topics that relate to computation are algorithm validation, complexity estimation and optimization. Wide part of theoretical computer science deals with these tasks. Complexity of tasks in general is examined studying the most relevant computational resources like execution time and space. The ranging of problems that are solvable with a given limited amount of time and space into well-defined classes is a very intricate task, but it can help incredibly to save time and money spent on the algorithms design. Vast collections of papers were dedicated to algorithm development. A short historical overview of the fundamental issues in theory of computation can be found in [1]. We do not discuss precise definition of algorithm and complexity. The interested reader can apply for the information to one of the fundamental books on theory of algorithms, e.g. [2], [3].

Modern problems tend to be very intricate and relate to analysis of large data sets. Even if an exact algorithm can be developed its time or space complexity may turn out unacceptable. But in reality it is often sufficient to find an approximate or partial solution. Such admission extends the set of techniques to cope with the problem. We discuss heuristic algorithms which suggest some approximations to the solution of optimization problems. In such problems the objective is to find the optimal of all possible solutions that minimizes or maximizes an objective function. The objective function is a function used to evaluate a quality of the generated solution. Many real-world issues are easily stated as optimization problems. The collection of all possible solutions for a given problem can be regarded as a search space, and optimization algorithms, in their turn, are often referred to as search algorithms.

In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**, **NP** standing for Nondeterministic Polynomial time) is a class of problems having two properties:

- Any given solution to the problem can be **verified** quickly (in polynomial time); the set of problems with this property is called **NP**.

- If the problem can be **solved** quickly (in polynomial time), then so can every problem in **NP**.

Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today. As a consequence, determining whether or not it is possible to solve these problems quickly is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. An expert programmer should be able to recognize an NP-complete problem so that he or she does not unknowingly waste time trying to solve a problem which so far has eluded generations of computer scientists. Instead, NP-complete problems are often addressed by using approximation algorithms in practice.

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.

Algorithms are at the heart of problem solving in scientific computing and computer science. Unfortunately many of the combinatorial problems that arise in a computational context are NP-hard, so that optimal solutions are unlikely to be found in polynomial time. How can we cope with this intractability? One approach is to design algorithms that find approximate solutions guaranteed to be within some factor of the quality of the optimal solution. More recently, in large-scale scientific computing, even polynomial time algorithms that find exact solutions are deemed too expensive to be practical, and one needs faster (nearly linear time) approximation algorithms. We will consider the design of approximation algorithms for various graph-theoretical and combinatorial problems that commonly arise in scientific computing and computational

biology. These include set covers (vertex covers in hyper graphs), matching, coloring, and multiple sequence alignments in computational biology.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- **Approximation:** Instead of searching for an optimal solution, search for an "almost" optimal one.
- **Randomization:** Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. See Monte Carlo method.
- **Restriction:** By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- **Parameterization:** Often there are fast algorithms if certain parameters of the input are fixed.
- **Heuristic:** An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

Approximate algorithms entail the interesting issue of quality estimation of the solutions they find. Taking into account that normally the optimal solution is unknown, this problem can be a real challenge involving strong mathematical analysis. In connection with the quality issue the goal of the heuristic algorithm is to find as good solution as possible for all instances of the problem. There are general heuristic strategies that are successfully applied to manifold problems.

In computer science, a *heuristic algorithm*, or simply a *heuristic*, is an algorithm that is able to produce an acceptable solution to a problem in many practical scenarios, but for which there is no formal proof of its correctness. Alternatively, it may be correct, but may not be proven to produce an optimal solution, or to use reasonable resources. Heuristics are typically used when there is no known method to find an optimal solution, under the given constraints (of time, space *etc.*) or at all.

Two fundamental goals in computer science are finding algorithms with provably good run times and with provably good or optimal solution quality. A *heuristic* is an algorithm that abandons one or both of these goals; for example, it usually finds pretty good solutions, but there is no proof the solutions could not get arbitrarily bad; or it

usually runs reasonably quickly, but there is no argument that this will always be the case.

The term **Heuristic** is used for algorithms, which find solutions among all possible ones, but they do not guarantee that the best will be found; therefore they may be considered as approximate and not accurate algorithms. These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. The method used from a heuristic algorithm is one of the known methods, such as greediness, but in order to be easy and fast the algorithm ignores or even suppresses some of the problem's demands.

1.2 Organization of Thesis

The thesis is organized as follows:

Chapter-2 It presents some essential information about algorithms and computational complexity.

Chapter-3 It Presents definitions of NP-problem and heuristics. Some intractable problems that could help to understand deeper importance of heuristics are also mentioned.

Chapter-4 It presents problem statement that could analyses the problem statement that we are going deal.

Chapter-5 It presents results & discussion.

Finally, the last chapter is devoted to the conclusion.

CHAPTER 2

ALGORITHMS AND COMPLEXITY

2.1 Introduction to Algorithms

Classes of time complexity are defined to distinguish problems according to their “hardness”[4].

Class P consists of all those problems that can be solved on a deterministic Turing machine in polynomial time from the size of the input. Turing machines are an abstraction that is used to formalize the notion of algorithm and computational complexity.

Class NP consists of all those problems whose solution can be found in polynomial time on a non-deterministic Turing machine. Since such a machine does not exist, practically it means that an exponential algorithm can be written for an NP problem, nothing is asserted whether a polynomial algorithm exists or not.

A subclass of NP, **class NP-complete** includes problems such that a polynomial algorithm for one of them could be transformed to polynomial algorithms for solving all other NP problems. Finally, the **class NP-hard** can be understood as the class of problems that are NP-complete or harder. NP-hard problems have the same trait as NP-complete problems but they do not necessarily belong to class NP, that is class NP-hard includes also problems for which no algorithms at all can be provided.

In order to justify application of some heuristic algorithm we prove that the problem belongs to the classes NP-complete or NP-hard. Most likely there are no polynomial algorithms to solve such problems; therefore, for sufficiently great inputs heuristics are developed.

2.2 Complexity Classes

Almost all the algorithms we have studied thus far have been **polynomial-time algorithms**. On inputs of size n , their worst-case running time is $O(n^k)$ for some constant k . It is natural to wonder whether all problems can be solved in polynomial-time. The answer is No.

For example, there are problems, such as Turing's famous problem "Halting problem", that can't be solved by any computer, no matter how much time is provided. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k .

Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require super polynomial time as being intractable, or hard. Generally there are four classes of problems based on time complexity. Classes of time complexity are sets of problems with the same bounds on execution time [4].

1. Class P
2. Class Exp
3. Class NP
 - Class NP-complete
 - Class NP-Hard

Class P: The class P consists of those problems that are solvable in polynomial – time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k , where n is of the input to the problem .

Example: Different Sorting and Searching Algorithms like

- Binary search
- Linear search
- Insertion sort
- Selection sort
- Bubble sort, etc...

Class Exp: Exp is the class of problems that are solvable by exponential time algorithms. Formally, Exp is the class of abstract problems, which have an algorithm that solve it in time $t(n)$ and $t(n) \in O(2^{nk})$ for some constant k .

Clearly the class P is contained in the class Exp, that is, Exp is a more general class. The problems contained in Exp but not in P, are generally considered hard problems.

Classes NP and Co-NP:

There are problems that only have known algorithmic solutions that grow in time at the rates of $O(n!)$, $O(2^n)$, and even $O(n^n)$. These algorithms are known as non-polynomial (or just NP) time algorithms.

The class NP consists of those problems that are “verifiable” in polynomial time. What we mean here is that if we are given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

NP is the class of decision problems that have a polynomial-time verification algorithm.

Co-NP is the class of decision problems for which the corresponding complement problem belongs to NP.

For instance, the problem “Is a given integer number a composite number?” belongs to the class NP and the problem “Is a given integer number a prime number?” belongs to the class Co-NP.

Needless to say, non-polynomial time solutions are to be avoided. Unfortunately, not all non-polynomial time solutions have known polynomial time equivalents. Worse, it is not possible to create a polynomial time solution for some problems. These problems and their algorithms are known non-polynomial time complete (or just NP-Complete).

Class NP-Complete: NP + polynomial algorithm for one of the problem can be transformed to solve all other NP problems in polynomial time.

We will state without proof that if any NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial time algorithm. Most theoretical computer scientists believe that the NP-complete problems are intractable.

One of the most accepted ways to prove that a problem is hard is to prove it NP-complete. If a decision problem is NP-complete we are almost certain that it cannot be solved optimally in polynomial time.

If we are given an algorithm to solve a problem, we can often use it to solve other similar problems. Given two problems A and B, we can specify in advance how to use any algorithm for problem B to solve problem A. Such a specification is called a reduction from problem A to B. If we are able to prove that the process is correct, it is said that we have reduced A to B.

In the 1970s and 1980s, a lot of decision problems for which the only accepted answers are either “Yes” or “No” were proved to be reducible to each other. All these problems have a common property: for every input to a problem with a “Yes” solution there is a proof that the input has solution “Yes” and this proof can be verified in polynomial

time. Any problem with this property is called an NP problem. We say this for decision problems, but that is easy to infer that all problems which can be solved by polynomial time algorithms satisfy this property. We say that $P \subseteq NP$ where P is the set of problems which can be solved by polynomial time algorithms and NP is the set of NP problems.

A problem which can be said to be “harder” than all NP problems, that is a problem to which every problem in NP be reduced, is called NP-hard. If an NP-hard problem is itself an NP problem it is called NP-complete. Thus all NP-complete problems are equally hard to solve, since they are inter-reducible. If there is a polynomial time algorithm for an NP-complete problem then $P = NP$ and every NP problem can be solved in polynomial time. Despite enormous efforts the question whether $P = NP$ is still unanswered. The common belief nowadays is that $P \subseteq NP$ and a big part of the research in Theoretical Computer Science has $P \subseteq NP$ as a fundamental assumption.

So according to the above discussion, we realize that there are some problems that cannot be approximated with efficient algorithms at all. On the other hand, it depends on the problem since we know that for some problems that are equally hard to solve optimally, some can be approximated very well with efficient algorithms. This makes it possible that some problems can be efficiently approximated using algorithms based on evolutionary techniques while others cannot [Mos01].

2.3 Examples of different NP-Complete problems

2.3.1 Boolean satisfiability problem (SAT)

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. To emphasize the binary nature of this problem, it is frequently referred to as **Boolean** or **propositional satisfiability**[5]. The shorthand “**SAT**” is also commonly used to denote it, with the implicit understanding that the function and its variables are all binary-valued.

Many (practical) applications:

- Formal methods:
 - Hardware model checking; Software model checking; Termination analysis of Term-rewrite systems; Test pattern generation (testing of software & hardware);
- Artificial intelligence:
 - Planning; Knowledge representation; Games (n-queens, sudoku, social golpher's,)
- Bioinformatics:
 - Haplotype inference; Pedigree checking; Maximum quartet consistency; etc.
- Design automation:
 - Equivalence checking; Delay computation; Fault diagnosis; Noise analysis.
- Security: Cryptanalysis; Inversion attacks on hash functions; etc.
- Computationally hard problems: Graph coloring; Traveling salesperson; etc.
- Mathematical problems: Vander Wardens numbers; etc

2.3.2 Fifteen puzzle problem

In the 1870's the impish puzzle maker Sam Loyd caused quite a stir in the United States, Britain, and Europe with his now-famous 15-puzzle. In its original form, the puzzle consists of fifteen square blocks numbered 1 through 15 but otherwise identical and a square tray large enough to accommodate 16 blocks. The 15 blocks are placed in the tray as shown in Figure 1, with the lower right corner left empty. A legal move consists of sliding a block adjacent to the empty space into the empty space. Thus, from the starting placement, block 12 or 15 may be slid into the empty space. The object of the puzzle is to use a sequence of legal moves to switch the positions of blocks 14 and 15 while returning all other blocks to their original positions [6].

The ***n*-puzzle** is known in various versions, including the **8 puzzle** [8], the **15 puzzle**, and with various names. It is a sliding puzzle that consists of a frame of numbered square tiles in random order with one tile missing. If the size is 3×3 , the puzzle is called the 8-puzzle or 9-puzzle, and if 4×4 , the puzzle is called the 15-puzzle or 16-puzzle. The object of the puzzle is to place the tiles in order (see diagram) by making sliding moves that use the empty space.

The n -puzzle is a classical problem for modeling algorithms involving heuristics. Commonly used heuristics for this problem include counting the number of misplaced tiles and finding the sum of the Manhattan distances between each block and its position in the goal configuration. Note that both are *admissible*, i.e., they never overestimate the number of moves left, which ensures optimality for certain search algorithms such as A*.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 2.1. The starting position for the 15-puzzle.

2.3.3 The Temporal Knapsack Problem

The **knapsack problem** is a problem in combinatorial optimization. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

A similar problem often appears in business, combinatory, complexity theory, cryptography and applied mathematics.

Definition:

In the following, we have n kinds of items, 1 through n . Each kind of item j has a value p_j and a weight w_j . We usually assume that all values and weights are nonnegative. The maximum weight that we can carry in the bag is W .

The **0-1 knapsack problem** [7] restricts the number x_j of copies of each kind of item to zero or one. Mathematically the 0-1-knapsack problem can be formulated as:

$$\begin{aligned}
 & \text{Maximize } \sum_{j=1}^n p_j x_j \\
 & \text{Subject to } \sum_{j=1}^n w_j x_j \leq W, \quad x_j \in \{0,1\}, \quad j = 1, \dots, n.
 \end{aligned}$$

The **bounded knapsack problem** restricts the number x_j of copies of each kind of item to a maximum integer value b_j . Mathematically the bounded knapsack problem can be formulated as:

$$\begin{aligned} & \text{Maximize } \sum_{j=1}^n p_j x_j \\ & \text{Subject to } \sum_{j=1}^n w_j x_j \leq W, \quad x_j \in \{0, 1, \dots, b_j\}, \quad j = 1, \dots, n. \end{aligned}$$

The unbounded knapsack problem places no upper bound on the number of copies of each kind item.

Different variations:

- Unbounded knapsack problem: no limits on the number of each item
- 0-1 knapsack problem: number of each item is either 0 or 1.
- Fractional knapsack problem: number of each item can be a fractional.

Solutions to general knapsack problem:

- Greedy algorithm approach
- Dynamic programming approach
- Genetic algorithm approach

2.3.4 Minesweeper

Over summer, Andy Miller and Sam Hazlehurst decided that they would realize their childhood dream of founding a game company. Their plan was to choose a niche so small that they couldn't fail to dominate it: Minesweeper games. When Yours & Mine Software finally shipped.

Minesweeper is easiest to learn by playing. For a good, if somewhat unstable, implementation, check out xmine. Minesweeper consists of a 2-dimensional grid of squares. Each square is originally covered; under some squares there are mines. When you **uncover** a square with a mine under it, you lose. When you **uncover** a square that doesn't have a mine under it, the game displays the number of mined squares adjacent to the current square (a number from 0 to 8). If this number is zero, the game must perform the **uncover** operation on all squares adjacent to it. A user **flags** a square to indicate that she thinks that there is a mine underneath the square. When all the squares on the board, which are not mines, have been uncovered, the player has Won the game.



Figure 2.2: The two pictures show what a Minesweeper game looks like at the beginning and at the end of a completed game.

The game has an 8x 8 [8] beginner level; a 16 x 16 intermediate level; and a 16 x 30 advanced level. The game also keeps track of the time that a player takes to complete a game, and uses these times to maintain three high-score lists (one for each level)[9]. High score lists consist of the ten highest scores and the players who achieved them.

2.3.5 Tetris

Tetris is a puzzle video game. When the game starts, only an empty board with borders drawn around its edges should be displayed. A Tetris piece, chosen randomly from the seven possible Tetris pieces shown below should appear at the top of the board. This piece should fall by moving down the board, one square at a time. A piece cannot fall into a square already occupied by a previously fallen piece. When a piece can fall no further, it should stop moving; a new random piece should then appear at the top of the board and begin to fall. As pieces fall, rows (or horizontal lines) of occupied squares spanning the board's width may form. When such a line is formed, it disappears and all the squares above it fall down one line to fill the newly empty row. This process continues until there is either a piece in the top row of the board or a new piece appears and has no room to fall because it is already resting on a previously fallen piece. The game is then over, and everything on the board should stop completely. A message should be displayed to let the user know that the game is over[10].

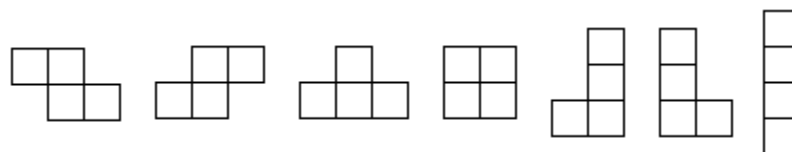


Figure 2.3: The seven basic Tetris pieces: each an arrangement of four connected squares.

While a piece is falling, the player may rotate or shift it by pressing certain keys on the key board. Pressing 'j' should shift the piece one square to the left. Pressing 'l' should shift the piece one square to the right. Pressing 'k' should rotate the piece counterclockwise by ninety degrees. At regular intervals, the piece should simply fall vertically one square at a time. The player should be able to drop the piece by pressing the space bar. By dropping a piece, the player forfeits his/her chance to manipulate the piece any further and the piece simply falls as far as it can. The player should be able to cause the piece to drop more quickly by pressing 'm'. The player should be able to pause the game at any time by pressing 'p'. When the game is paused, the player should not be able to manipulate the piece in any way. Pressing 'p' again should allow the user to resume play. When the game is paused, some sort of notification should be given to the user, most likely using either a JLabel or using your Graphics2D to paint some Color Text.



Figure 2.4: *Tetris* on the Nintendo Game Boy system, which uses the original randomizer.

Likewise, when there is a game over, the user should receive some sort of notification, and no pieces should be allowed to be manipulated.

2.3.6 Hamiltonian cycle problem

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). Both problems are NP-complete. The problem of finding a Hamiltonian cycle or path is in FNP.

There is a simple relation between the two problems. 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 .

Hamiltonian Cycle is in NP because a given list of vertices can be checked for a tour in polynomial time. Now, we want to show that 3-SAT reduces to Hamiltonian Cycle in polynomial time. Thus, we need some way of mapping the variables and clauses to a Hamiltonian cycle problem graph [11].

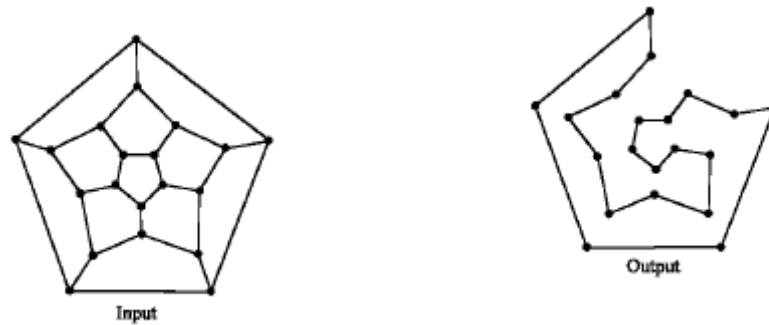


Figure 2.5: The mapping of Hamilton Cycle Problem Graph.

While this graph is complex at first, look at how it is built. For each variable, there is a row of at most $2n$ nodes; where n is the number of clauses. Two of these nodes are connected to each clause if the variable appears in that clause. The ends of each row of nodes are connected to the ends of the next. Walking left to right down a row means “true,” while right to left means “false.” In this way, a Hamiltonian cycle through the graph represents an assignment of truth values to the variables in a list of 3-SAT clauses.

2.3.7 Traveling salesman problem(TSP)

The Travelling Salesman Problem (TSP) is a simple combinatorial problem [12]. It can be stated Very simply:

- A salesman spends his time visiting n cities (or nodes) . In one tour he visits each city just Once, and finishes up where he started. In what order should he visit them to minimize the distance traveled?
- There exists an edge between every pair of cities expressing the distance between the two Corresponding cities.
- If there are only 2 cities then the problem is trivial, since only one tour is possible. In either case the number of solutions becomes extremely large for large n , so that an exhaustive search is impractical.

Over the past 15 years, the record of for the largest nontrivial TSP instance solved to optimality has increased from 318 cities (1980) to 2392 cities (1987) to 7397 cities(1994).

The last result has taken 3-4 years on a network of machines.

2.3.8 Subset sum problem

In computer science, the subset sum problem is an important problem in complexity theory and cryptography. The problem is this: given a set of integers, does the sum of some non-empty subset equal exactly zero? For example, given the set $\{-7, -3, -2, 5, 8\}$, the answer is YES because the subset $\{-3, -2, 5\}$ sums to zero. The problem is NP-Complete.

Definition: A *knapsack vector* is a set A of positive and pairwise different integers such that

$$A = (a_1, a_2, \dots, a_n), \quad a_i \neq a_j \text{ if } i \neq j$$

The *subset sum problem* is the following: Given

A knapsack vector $A = (a_1, a_2, \dots, a_n)$ and a positive integer s , called the sum

The question is then

Is there a subset A' of A with $\sum_{a' \in A'} (a') = s$, or equivalent:

Does there exist a vector $X = (x_1, x_2, \dots, x_n)$, $x_i \in \{0, 1\}$, with $AX = s$?

Where AX means $a_1x_1 + a_2x_2 + \dots + a_nx_n$.

The problem just described is a decision problem. If a solution is desired, the problem becomes a functional problem: the task is to compute a solution vector $X = (x_1, x_2, \dots, x_n)$, $x_i \in \{0, 1\}$ such that $AX = s$, provided a solution exists.

The functional problem of a knapsack vector A together with a sum s is denoted by (A, s) . **Example:**

Let A be the set $(15, 22, 14, 26, 32, 9, 16, 8)$ and let s be 53. Then a solution to $(A, 53)$ is given by the vector $X = (1, 1, 0, 0, 0, 0, 1, 0)$, because $AX = a_1 + a_2 + a_7 = 15 + 22 + 16 = 53$. The corresponding subset is $A' = (15, 22, 16)$.

In general there may exist several solutions to (A, s) as in this example. Another solution for $(A, 53)$ is $X = (0, 1, 1, 0, 0, 1, 0, 1)$ with the subset $A' = (22, 14, 9, 8)$.

No polynomial algorithm is known solving the general subset sum problem. The subset sum problem is in the complexity class NP. The decision problem is NP-complete and the corresponding functional problem is NP-hard. The decision problem and the

functional problem are equivalent with respect to the complexity meaning if a polynomial algorithm is known solving the decision problem, this algorithm can also be used for solving the functional problem and vice versa.

2.3.9 Subgraph isomorphism problem

Mathematical motivation

- NP-complete problems are a challenge to theoretical computer science

Non-mathematical motivation

- Pattern recognition and computer vision
- Computer-aided design
- Image processing
- Graph grammars and graph transformation
- Biocomputing

Subgraph isomorphism is an important and very general form of *exact* pattern matching

- String searching
- Sequence alignment
- Tree comparison
- Pattern matching on graphs

Example. ***Tree isomorphism***

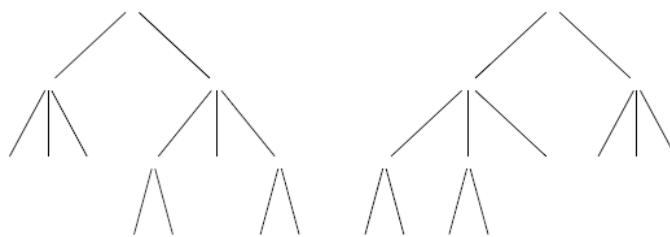


Figure 2.6: Shows Isometric Trees

Example. *Graph isomorphism*

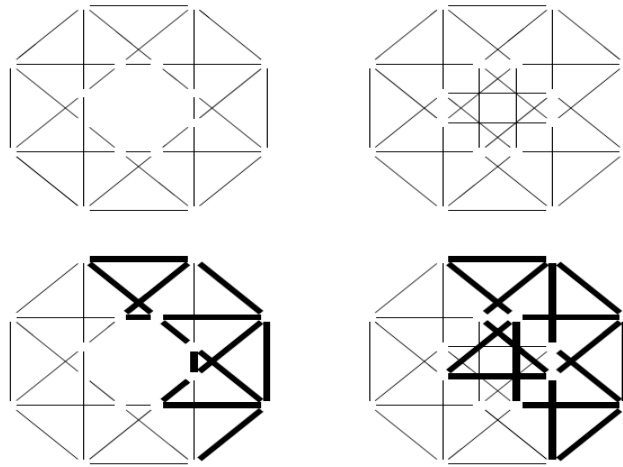


Figure 2.7: Shows Isometric Graphs

2.3.10 Clique Problem

In computational complexity theory, the **clique problem** is a graph-theoretic NP-complete problem. The problem was one of Richard Karp's original 21 problems shown NP-complete in his 1972 paper "Reducibility among Combinatorial Problems". This problem was also mentioned in Cook's paper introducing the theory of NP-complete problems. A clique in a graph is a set of pairwise adjacent vertices, or in other words, an induced subgraph which is a complete graph. In the graph at the right, vertices 1, 2 and 5 form a clique, because each has an edge to all the others.

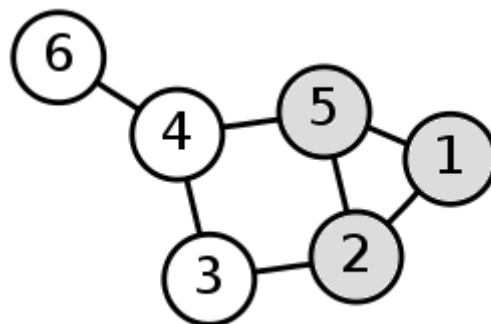


Figure 2.8: Shows a Clique Problem of size 3

Then, the clique problem is the problem of determining whether a graph contains a clique of at least a given size k . Once we have located k or more vertices which form a clique, it's trivial to verify that they do, which is why the clique problem is in NP. The

corresponding optimization problem, the **maximum clique problem**, is to find the largest clique in a graph.

2.3.11 Vertex Cover Problem

In computer science, the **vertex cover problem** or **node cover problem** is an NP-complete problem and was one of Karp's 21 NP-complete problems. It is often used in complexity theory to prove NP-hardness of more complicated problems.

Definition: A vertex cover for an undirected graph $G = (V, E)$ is a subset S of its vertices such that each edge has at least one endpoint in S . In other words, for each edge ab in E , one of a or b must be an element of S .

Example: In the graph on the right, $\{1,3,5,6\}$ is an example of a vertex cover of size 4. However, it is not a smallest vertex cover since there exist vertex covers of size 3, such as $\{2,4,5\}$ and $\{1,2,4\}$.

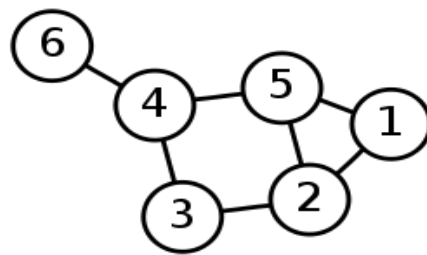


Figure 2.9: Shows different vertex vectors.

The **vertex cover problem** is the optimization problem of finding a smallest vertex cover in a given graph. INSTANCE: Given a graph G

OUTPUT: Smallest number k such that there is a vertex cover S for G of size k .

Equivalently, the problem can be stated as a decision problem:

INSTANCE: Graph G and positive integer k .

QUESTION: Is there a vertex cover S for G of size at most k ?

Vertex cover is closely related to the Independent Set problem. A set of vertices S is a vertex cover if and only if its complement $\bar{S} = V \setminus S$ is an independent set. It follows that a graph with n vertices has a vertex cover of size k if and only if the graph has an independent set of size $n - k$. In this sense, the two problems are dual to each other.

2.3.12 Independent Set Problem

In mathematics, the **Independent Set Problem (IS)** is a well-known problem in graph theory and combinatorics. The independent set problem is known to be NP-complete. It is almost identical to the **Clique Problem**.

Given a graph G , an independent set is a subset of its vertices that are pairwise not adjacent. In other words, the subgraph induced by these vertices has no edges, only isolated vertices. Then, the independent set problem asks: given a graph G and a positive integer k , does G have an independent set of cardinality at least k ?

The corresponding optimization problem is the **maximum independent set problem**, which attempts to find the largest independent set in a graph. Given a solution to the decision problem, binary search can be used to solve the optimization problem with $O(\log |V|)$ invocations of the decision problem's solution. The optimization problem is known to have no constant-factor approximation algorithm if $P \neq NP$. Independent set problems and clique problems may be easily translated into each other: an independent set in a graph G is a clique in the complement graph of G , and vice versa.

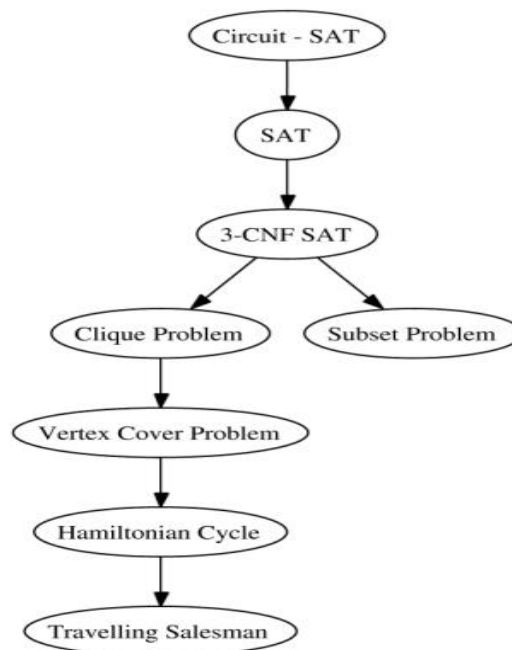


Figure 2.10: Shows the relation between different NP-problems

2.4 Hierarchy of complexity classes

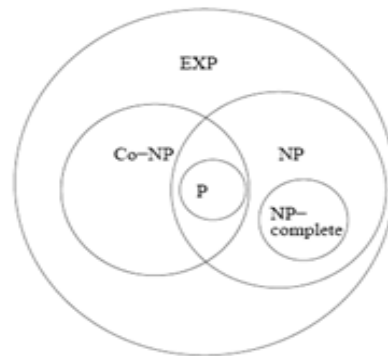


Figure 2.11: Shows the known relationships between the complexity classes mentioned above. The following are the major unsolved questions:

- $P=NP?$
- $NP=Co-NP?$
- $P=NP \setminus Co-NP?$

2.5 NP-Problems in different fields

- **Aerospace engineering:** optimal mesh partitioning for finite elements.
- **Biology:** protein folding.
- **Chemical engineering:** heat exchanger network synthesis.
- **Civil engineering:** equilibrium of urban traffic flow.
- **Economics:** computation of arbitrage in financial markets with friction.
- **Electrical engineering:** VLSI layout.
- **Environmental engineering:** optimal placement of contaminant sensors.
- **Financial engineering:** find minimum risk portfolio of given return.
- **Game theory:** find Nash equilibrium that maximizes social welfare.
- **Genomics:** phylogeny reconstruction.
- **Mechanical engineering:** structure of turbulence in sheared flows.
- **Medicine:** reconstructing 3-D shape from biplane angiogram.
- **Operations research:** optimal resource allocation.
- **Physics:** partition function of 3-D Ising model in statistical mechanics.
- **Politics:** Shapley-Shubik voting power.
- **Pop culture:** Minesweeper consistency.

CHAPTER 3

HEURISTIC ALGORITHMS

3.1 What is Heuristic?

A heuristic algorithm is an algorithm that using a strategy that does not examine all possible solutions to a problem. Heuristic algorithms make no attempt to find the perfect solution to the problem. Instead, heuristic algorithms look for a "good enough" solution in an acceptable amount of time. A heuristic algorithm is one that will provide a solution close to the optimal, but may or may not be optimal. The concept of heuristic solutions to problems normally solved via non-polynomial time algorithms has changed the way programmers regard NP and NP-Complete problems.

In computer science, a **Heuristic Algorithm** or simply a **Heuristic** is an algorithm that ignores whether the solution to the problem can be proven to be correct, but which usually produces a good solution or solves a simpler problem that contains or intersects with the solution of the more complex problem. Heuristics are typically used when there is no known way to find an optimal solution, or when it is desirable to give up finding the optimal solution for an improvement in run time.

Two fundamental goals in computer science are finding algorithms with provably good run times and with provably good or optimal solution quality. A **Heuristic** is an algorithm that abandons one or both of these goals; for example, it usually finds pretty good solutions, but there is no proof the solutions could not get arbitrarily bad; or it usually runs reasonably quickly, but there is no argument that this will always be the case.

For instance, say you are packing odd-shaped items into a box. Finding a perfect solution is a hard problem: there is essentially no way to do it without trying every possible way of packing them. What most people do, then, is "put the largest items in first, then fit the smaller items into the spaces left around them." This will not necessarily be perfect packing, but it will usually give packing that is pretty good. It is an example of a heuristic solution.

The principal advantages of heuristic algorithms are that such algorithms are (often) *conceptually simpler* and (almost always) much *cheaper computationally* than optimal algorithms.

Example: **Assignment Problem**

Consider the following problem: 3 men are to be assigned to 3 jobs - where the assignment cost is given by the matrix below:

		Job		
		1	2	3
Man	A	1	3	4
	B	3	7	4
	C	3	4	2

Only one man can be assigned to one job and all the men should be assigned. What would be a heuristic algorithm for this problem?

We should stress here that a heuristic algorithm should be capable of being applied to the problem even if the costs in the above matrix are changed (i.e. a heuristic algorithm is a set of general rules for solving the problem that are independent of the particular data case being considered).

Heuristic for the Assignment Problem: One (simple) heuristic for the assignment problem would be: choose a man and a job at random. Assign the chosen man to the chosen job. Delete the chosen man and the chosen job from the problem and repeat with this new (smaller) problem.

This heuristic does not use any of the cost information and so we would not expect it to give very good results.

Note however the idea of *repetition*. This is a common concept in heuristic algorithms (both because it eases the task of programming the heuristic, and because if a certain algorithmic step is a good idea then why not repeats it?).

A better heuristic might be: choose the smallest cost in the cost matrix (ties broken arbitrarily) and assign the corresponding man to the corresponding job - delete them from the problem and repeat with this new (smaller) problem. This heuristic would give the solution:

Cost 1 Assign A to job 1

Cost 2 Assign C to job 3

Cost 7 Assign B to job 2

Total cost 10

This illustrates a problem that often occurs with heuristics in that by the third assignment (of B to job 2) we have been "painted into a corner" by previous assignments and have little or no choice left (with the result that we have to assign B to job 2 at relatively high cost). Because of this problem a common idea with heuristics is the concept of *interchange* - the basic idea here is to juggle with the current solution to see if we can improve it e.g. with the solution above could we improve it by, for example, swapping the assignments of A and C thereby assigning A to job 3 and C to job 1? Here this swap is not worthwhile but some swaps (interchanges) are e.g. swap the assignments of A and B.

Note particularly here that we designed the above heuristic without ever having a mathematical formulation of the problem. It is difficult to imagine the variety of existing computational tasks and the number of algorithms developed to solve them. Algorithms that either give nearly the right answer or provide a solution not for all instances of the problem are called heuristic algorithms. This group includes a plentiful spectrum of methods based on traditional techniques as well as specific ones.

3.2 Different Heuristic Strategies

3.2.1 Exhaustive search: The simplest of search algorithms is **exhaustive search** that tries all possible solutions from a predetermined set and subsequently picks the best one. Many problems have complexity worse than polynomial, such as the recursive Towers of Hanoi, which is $O(2^n)$. Some are inherently expensive, such as finding all permutations of a string of n characters, which is $O(n!)$. Faced with an expensive algorithm, one should look for techniques to reduce the work done. We shall consider this with the example of a game-playing strategy that uses a search tree to look ahead some number of moves to determine the best move.

Example: A) Depth-First Search

- Always explore first child next
- Evaluate other children before root
- Use stack as data structure to hold pending nodes (or use recursive code)
- Goes deep into search tree quickly
- If tree has infinite levels, DFS may not terminate
- Aggressive attack, good if finite depth and multiple solutions

Order of Evaluation of Nodes in Search Tree

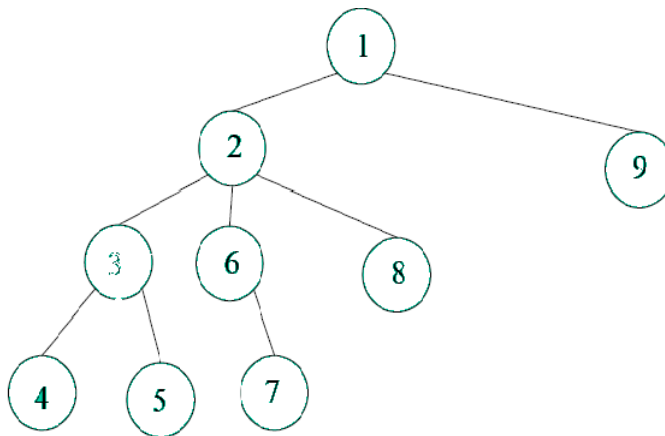


Figure 3.1: Shows order of evaluation of nodes in DFS

B) Breadth-First Search

Explore all nodes at given level before descending to next

- Use queue as data structure
- BFS is slower but safer than DFS, especially if there is no guarantee on number of levels in search tree

Order of Evaluation of Nodes in Search Tree

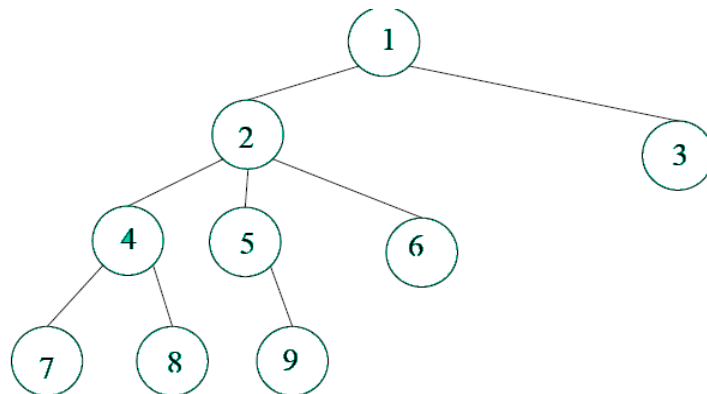


Figure 3.2: Shows order of evolution of nodes in BFS

3.2.2 Local Search

Local Search is a version of exhaustive search that only focuses on a limited area of the search space. Local search can be organized in different ways. Popular hill-climbing techniques belong to this class. Such algorithms consistently replace the current solution with the best of its neighbours if it is better than the current.

In **Hill Climbing** the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move if town B or C appear nearer to town D than town A does. In *steepest ascent* hill climbing you will always make your next state the best successor of your current state, and will only make a move if that successor is better than your current state.

This can be described as follows:

- Start with *current-state* = initial-state.
- Until *current-state* = goal-state OR there is no change in *current-state* do:
- Get the successors of the current state and use the evaluation function to assign a score to each successor.
- If one of the successors has a better score than the current-state then set the new current-state to be the successor with the best score.

Note that the algorithm does not attempt to exhaustively try every node and path, so no node list or agenda is maintained - just the current state. If there are loops in the search space then using hill climbing you shouldn't encounter them - you can't keep going up and still get back to where you were before. Hill climbing terminates when there are no successors of the current state, which are better than the current state itself.

3.2.3 Divide and Conquer

Divide and Conquer algorithms try to split a problem into smaller problems that are easier to solve. Solutions of the small problems must be combinable to a solution for the original one. This technique is effective but its use is limited because there is no a great number of problems that can be easily partitioned and combined in a such way.

Examles:

- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Stassen's algorithm, VLSI tree layout, etc.....

3.2.4 Branch-and-Bound

Branch-and-Bound technique is a critical enumeration of the search space. It enumerates, but constantly tries to rule out parts of the search space that cannot contain the best solution.

Example: 4-Queens Problem

FIFO branch-and-bound algorithm

- Initially, there is only one live node; no queen has been placed on the chessboard
- The only live node becomes E-node
- Expand and generate all its children; children being a queen in column 1, 2, 3, and 4 of row 1 (only live nodes left)
- Next E-node is the node with queen in row 1 and column 1
- Expand this node, and add the possible nodes to the queue of live nodes
- Bound the nodes that become dead nodes

Compare with backtracking algorithm

- Backtracking is superior method for this search problem

Where

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E- node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node

3.2.5 Dynamic Programming

Dynamic Programming is an exhaustive search that avoids re-computation by storing the solutions of sub problems. The key point for using this technique is formulating the solution process as a recursion. Dynamic Programming is a recursive method for solving *sequential decision problems* (hereafter abbreviated as SDP). Also known as *backward induction*, it is used to find *optimal decision rules* in “games against nature” and *subgame perfect equilibria* of dynamic multi-agent games, and competitive equilibria in dynamic economic models. Dynamic programming has enabled economists to formulate and solve a huge variety of problems involving sequential

decision making under uncertainty, and as a result it is now widely regarded as the single most important tool in economics.

Examples: a). Longest Common Subsequence (LCS)

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, find a longest subsequence common to them both.

b). Optimal Substructure

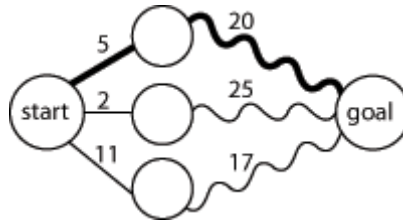


Figure 3.3

Figure 3.3. Finding the shortest path in a graph using optimal substructure; a straight line indicates a single edge; a wavy line indicates a shortest path between the two vertices it connects (other nodes on these paths are not shown); the bold line is the overall shortest path from start to goal.

c). Overlapping Sub problems (Fibonacci sequence)

3.2.6 Greedy Technique

A popular method to construct successively space of solutions is *greedy* technique, that is based on the evident principle of taking the (local) best choice at each stage of the algorithm in order to find the global optimum of some objective function.

A technique used in solving optimization problems. Typically, we are given a set of n inputs and the goal is to find a subset (or some output) that satisfies some constraints. Any subset (or output) that satisfies these constraints is called a feasible solution. In an optimization problem, we need to find a feasible solution that maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution. The greedy technique works in stages, considering one input at a time (typically in some clever order). At each stage, a decision is made depending on whether it is best at this stage. For example, a simple criterion can be whether adding the current input will lead to an infeasible solution or not. Thus, a locally optimal choice is made in the hope that it will lead to a globally optimal solution.

Basic steps to finding efficient greedy algorithms:

- Start by finding a dynamic programming style solution
- Prove that at each step of the recursion, the min/max can be satisfied by a “greedy choice” (*greedy substructure*)
- Show that only one recursive call needs to be made once the greedy choice is assumed. This is often natural when all the recursive calls are made by the min/max.
- Find the recursive solution using the greedy choice
- Convert to an iterative algorithm if possible

More generally, taking the direct approach:

- Show the problem is reduced to a sub problem via a greedy choice
- Prove there is an optimal solution containing the greedy choice
- Prove that combining the greedy choice with an optimal solution for the remaining sub problem yields an optimal solution

Example:

- For the MST problem: Prim’s and Kruskal’s algorithms
- For the SSSP problem: Dijkstra’s algorithm

Remember, Dijkstra only works for graphs with no negative edge weights.

Usually heuristic algorithms are used for problems that cannot be easily solved.

3.3 Heuristic Techniques

Branch-and-bound technique and **dynamic programming** are quite effective but their time-complexity often is too high and unacceptable for NP-complete tasks.

3.3.1 Hill-Climbing

Hill-climbing algorithm is effective, but it has a significant drawback called **premature convergence**. Since it is “greedy”, it always finds the nearest local optima of low quality. The goal of modern heuristics is to overcome this disadvantage.

Premature convergence: When a genetic algorithms population converges to something, which is not the solution, we wanted.

3.3.2 Simulated Annealing

Simulated annealing algorithm [13], invented in 1983, uses an approach similar to hill climbing, but occasionally accepts solutions that are worse than the current. The probability of such acceptance is decreasing with time.

Simulated annealing (SA) is a generic probabilistic meta-algorithm for the global optimization problem, namely locating a good approximation to the global minimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more effective than exhaustive enumeration — provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

Some Real Applications of Simulated Annealing:

- Determining the sequence of observations for an automated astronomical telescope.
- Computer Aided Geometric Design.
- Optimization of Econometric Statistical Functions.
- Games with random moves determined by the simulated annealing algorithm.
- Arranging connections on chips and switching devices in telephone networks.

The SA algorithm: In the Simulated Annealing algorithm, an objective function to be minimized is defined. Here it will be the total path length through a set of points. The distance between each pair of points is equivalent to the "energy" of a molecule. Then, "temperature" is the average of these lengths. Starting from an initial point, the algorithm swaps a pair of points and the total "energy" of the path is calculated.

3.3.3 Tabu Search

Tabu search [14] extends the idea to avoid local optima by using memory structures. The problem of simulated annealing is that after “jump” the algorithm can simply repeat its own track. Tabu search prohibits the repetition of moves that have been made recently.

Webster’s dictionary defines the word *tabu* or *taboo* to mean “banned on grounds of morality or taste or as constituting a risk...”. Tabu Search (TS) is an optimization method designed to help a search negotiate difficult regions (i.e. to escape from local minima or to cross-infeasible regions of the search space) by imposing restrictions. It was originally developed as a method for solving combinatorial optimization problems (these are problems where the control variables are some form of ordered list — the *Travelling Salesman Problem* is the classic example), but variants to solve continuous and integer optimization problems have also been developed.

Example: ***The Classical Vehicle Routing Problem:***

Vehicle Routing Problems have very important applications in the area of distribution management. As a consequence, they have become some of the most studied problems in the combinatorial optimization literature and large number of papers and books deal with the numerous procedures that have been proposed to solve them. These include several TS implementations that currently rank among the most effective. The *Classical Vehicle Routing Problem* (CVRP) is the basic variant in that class of problems. It can formally be defined as follows. Let $G = (V, A)$ be a graph where V is the vertex set and A is the arc set. One of the vertices represents the *depot* at which a fleet of m identical vehicles of capacity Q is based, and the other vertices customers that need to be serviced. With each customer vertex v_i are associated a demand q_i and a service time t_i . With each arc (v_i, v_j) of A are associated a cost c_{ij} and a travel time t_{ij} . The CVRP consists in finding a set of routes such that:

- Each route begins and ends at the depot;
- Each customer is visited exactly once by exactly one route;
- The total demand of the customers assigned to each route does not exceed Q ;
- The total duration of each route (including travel and service times) does not exceed a specified value L ;
- The total cost of the routes is minimized.

A feasible solution for the problem thus consists in a partition of the customers into m groups, each of total demand no larger than Q , that are sequenced to yield routes (starting and ending at the depot) of duration no larger than L .

Scheduling ➤ Flow-Time Cell Manufacturing ➤ Heterogeneous Processor Scheduling ➤ Workforce Planning ➤ Classroom Scheduling ➤ Machine Scheduling ➤ Flow Shop Scheduling ➤ Job Shop Scheduling ➤ Sequencing and Batching Design ➤ Computer-Aided Design ➤ Fault Tolerant Networks ➤ Transport Network Design ➤ Architectural Space Planning ➤ Diagram Coherency ➤ Fixed Charge Network Design ➤ Irregular Cutting Problems Location and Allocation ➤ Multicommodity Location/Allocation ➤ Quadratic Assignment ➤ Quadratic Semi-Assignment ➤ Multilevel Generalized Assignment ➤ Lay-Out Planning ➤ Off-Shore Oil Exploration Logic and Artificial Intelligence ➤ Maximum Satisfiability ➤ Probabilistic Logic ➤ Clustering ➤ Pattern Recognition/Classification ➤ Data Integrity ➤ Neural Network Training and Design Technology ➤ Seismic Inversion ➤ Electrical Power Distribution ➤ Engineering Structural Design ➤ Minimum Volume Ellipsoids ➤ Space Station Construction ➤ Circuit Cell Placement	Telecommunications ➤ Call Routing ➤ Bandwidth Packing ➤ Hub Facility Location ➤ Path Assignment ➤ Network Design for Services ➤ Customer Discount Planning ➤ Failure Immune Architecture ➤ Synchronous Optical Networks Production, Inventory and Investment ➤ Flexible Manufacturing ➤ Just-in-Time Production ➤ Capacitated MRP ➤ Part Selection ➤ Multi-item Inventory Planning ➤ Volume Discount Acquisition ➤ Fixed Mix Investment Routing ➤ Vehicle Routing ➤ Capacitated Routing ➤ Time Window Routing ➤ Multi-Mode Routing ➤ Mixed Fleet Routing ➤ Traveling Salesman ➤ Traveling Purchaser Graph Optimization ➤ Graph Partitioning ➤ Graph Coloring ➤ Clique Partitioning ➤ Maximum Clique Problems ➤ Maximum Planner Graphs ➤ P-Median Problems General Combinational Optimization ➤ Zero-One Programming ➤ Fixed Charge Optimization ➤ Nonconvex Nonlinear Programming ➤ All-or-None Networks ➤ Bilevel Programming ➤ General Mixed Integer Optimization
---	--

Table 3.1 Illustrative Tabu Search Applications

3.3.4 Swarm Intelligence

Swarm Intelligence was introduced in 1989. It is an artificial intelligence technique, based on the study of collective behavior in decentralized, self-organized, systems. Two of the most successful types of this approach are Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO). In ACO artificial ants build solutions by moving on the problem graph and changing it in such a way that future ants can build better solutions. PSO deals with problems in which a best solution can be represented as a point or surface in an n-dimensional space. The main advantage of swarm intelligence [15] techniques is that they are impressively resistant to the local optima problem.

The typical swarm intelligence system has the following properties:

- It is composed of many individuals;
- The individuals are relatively homogeneous (i.e., they are either all identical or they belong to a few typologies);
- The interactions among the individuals are based on simple behavioral rules that exploit only local information that the individuals exchange directly or via the environment (stigmergy);
- The overall behavior of the system results from the interactions of individuals with each other and with their environment, that is, the group behavior self-organizes.

The characterizing property of a swarm intelligence system is its ability to act in a coordinated way without the presence of a coordinator or of an external controller. Many examples can be observed in nature of swarms that perform some collective behavior without any individual controlling the group, or being aware of the overall group behavior. Notwithstanding the lack of individuals in charge of the group, the swarm as a whole can show an intelligent behavior. This is the result of the interaction of spatially neighboring individuals that act on the basis of simple rules.

Few examples of scientific and engineering swarm intelligence studies.

- Clustering Behavior of Ants
- Nest Building Behavior of Wasps and Termites
- Flocking and Schooling in Birds and Fish
- Ant Colony Optimization
- Particle Swarm Optimization
- Swarm-based Network Management
- Cooperative Behavior in Swarms of Robots

3.3.5 Evolutionary Algorithms

Evolutionary Algorithms succeed in tackling premature convergence by considering a number of solutions simultaneously. In artificial intelligence, an **Evolutionary Algorithm** (EA) is a subset of evolutionary computation, a generic population-based meta heuristic optimization algorithm. An EA uses some mechanisms inspired by biological evolution: reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the environment within which the solutions "live" (see also cost function). Evolution of the population then takes place after the repeated application of the above operators. *Artificial evolution* (AE) describes a process involving individual *evolutionary algorithms*; EAs are individual components that participate in an AE.

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape; this generality is shown by successes in fields as diverse as engineering, art, biology, economics, marketing, genetics, operations research, robotics, social sciences, physics, politics and chemistry.

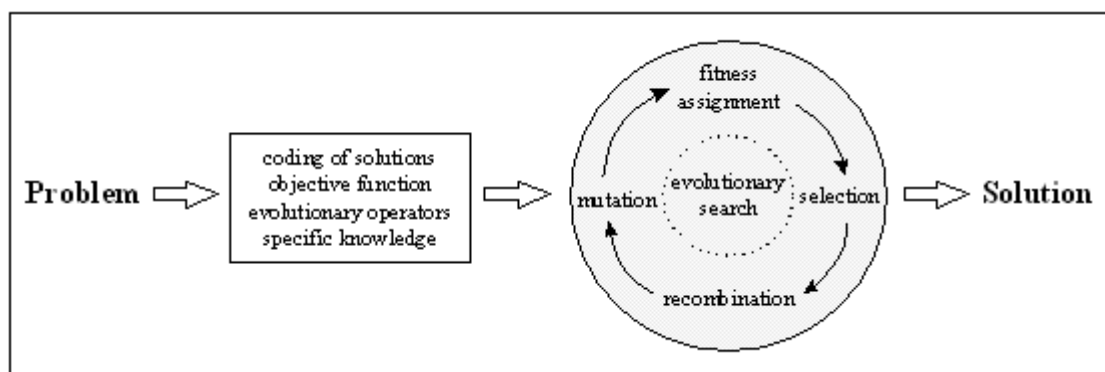


Figure 3.4: The process of Evolution Algorithms

Components of Evolutionary Algorithms: EA's have a number of components, procedures or operations that must be specified in order to define a particular EA [16]. The most important components are

- Representation (definition of individuals)
- Evaluation function (or fitness function)

- Population
- Parent selection mechanism
- Variation operators, Recombination and Mutation
- Survivor selection mechanism (Replacement)

Furthermore, to obtain a running algorithm the initialization procedure and termination condition must also be defined.

Types of Evolutionary Algorithms

- Genetic algorithms
- Evolutionary programming
- Evolution strategies
- Genetic programming
- Learning classifier systems

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural Genetics. They are based on the Darwin's theory of evolution which stressed the fact that the existence of all living things is based on the rule of 'survival of the fittest.' Darwin also postulated that new breeds or classes living things come into existence through the process of reproduction, crossover and mutation among existing organisms.

Genetic algorithms are substantially different to the more traditional search and optimization techniques. The main differences are:

- Genetic algorithms search a population of points in parallel, not from a single point.
- Genetic algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the direction of the search.
- Genetic algorithms use probabilistic transition rules, not deterministic rules.
- Genetic algorithms work on an encoding of a parameter set not the parameter set itself (except where real-valued individuals are used).
- Genetic algorithms may provide a number of potential solutions to a given problem and the choice of the final is left up to the user.

Pseudo-code for a genetic algorithm is as follows:

- *Initialize the population*
- *Evaluate initial population*
- *Repeat*
 - Perform competitive selection*
 - Apply genetic operators to generate new solutions*
 - Evaluate solutions in the population*
- *Until some convergence criteria is satisfied*

An extended discussion of issues involved with the implementation and use of evolutionary algorithms is included. Several different types of evolutionary search methods were developed independently. These include (a) genetic programming (GP), which evolve programs, (b) evolutionary programming (EP), which focuses on optimizing continuous functions without recombination, (c) evolutionary strategies (ES), which focuses on optimizing continuous functions with recombination, and (d) genetic algorithms (GAs), which focuses on optimizing general combinatorial problems.

3.3.6 Neural Networks

Neural Networks are inspired by biological neuron systems. They consist of units, called neurons, and interconnections between them. After special training on some given data set Neural Networks can make predictions for cases that are not in the training set. In practice Neural Networks do not always work well because they suffer greatly from problems of ***underfitting*** and ***overfitting***[17]. These problems correlate with the accuracy of prediction. If a network is not complex enough it may simplify the laws, which the data obey. From the other point of view, if a network is too complex it can take into account the noise that usually assists at the training data set while inferring the laws. The quality of prediction after training is deteriorated in both cases. The problem of ***premature convergence*** is also critical for Neural Networks.

The building blocks of neural networks

Neural networks are made of basic units (see Figure 18) arranged in layers. A unit collects information provided by other units (or by the external world) to which it is connected with weighted connections called synapses. These weights, called synaptic weights multiply (i.e., amplify or attenuate) the input information:

A positive weight is considered excitatory, a negative weight inhibitory.

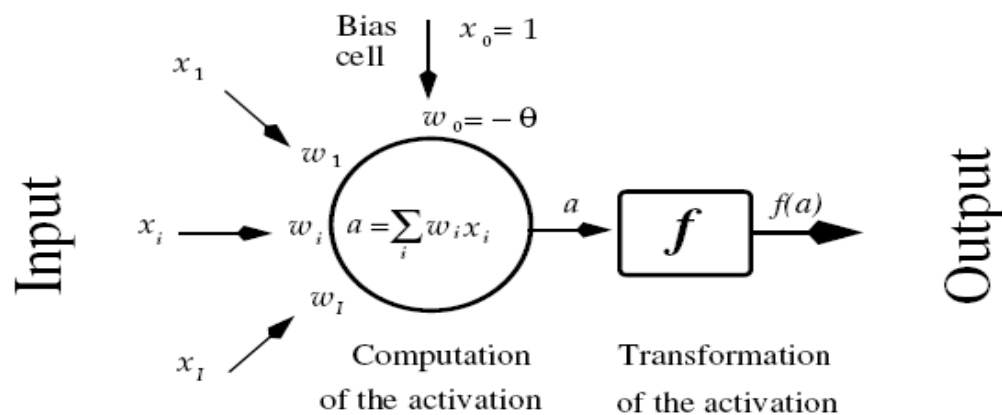


Figure 3.5: The Basic Neural Unit

Neural Network Components:

- Input Layer
- Hidden Layer
- Output layer
- Connections /Arcs
- Weights
- Activation functions
- Training set
- Learning

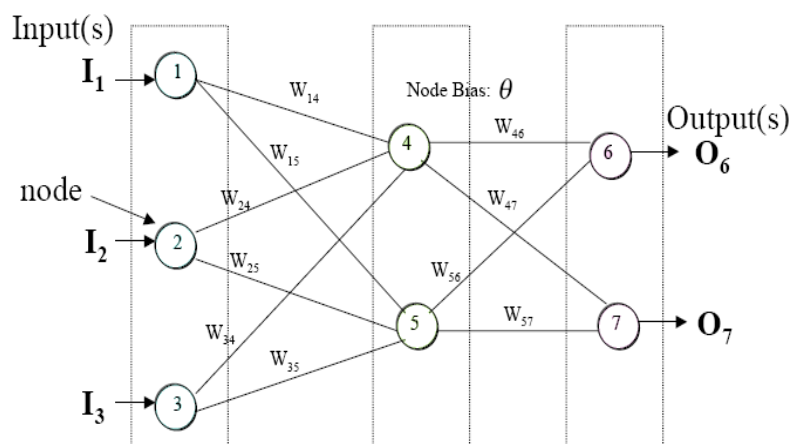


Figure 3.6: The Basic Components of Neural Networks

3.3.7 Support Vector Machines

Support Vector Machines (SVMs) extend the ideas of Neural Networks. They successfully overcome *premature convergence* since convex objective function is used, therefore, only one optimum exists. Classical divide and conquer technique gives elegant solution for separable problems. In connection with SVMs, that provide effective classification, it becomes an extremely powerful instrument.

A Support Vector Machine (SVM) performs classification by constructing an N -dimensional hyper plane that optimally separates the data into two categories. SVM models are closely related to neural networks. In fact, a SVM model using a sigmoid kernel function is equivalent to a two-layer, perception neural network.

Support Vector Machine (SVM) [18] models are a close cousin to classical multilayer perception neural networks. Using a kernel function, SVM's are an alternative training method for polynomial, radial basis function and multi-layer perception classifiers in which the weights of the network are found by solving a quadratic programming problem with linear constraints, rather than by solving a non-convex, unconstrained minimization problem as in standard neural network training.

In the parlance of SVM literature, a predictor variable is called an *attribute*, and a transformed attribute that is used to define the hyper plane is called a *feature*. The task of choosing the most suitable representation is known as *feature selection*. A set of features that describes one case (i.e., a row of predictor values) is called a *vector*. So the goal of SVM modeling is to find the optimal hyper plane that separates clusters of vector in such a way that cases with one category of the target variable are on one side of the plane and cases with the other category are on the other side of the plane. The vectors near the hyper plane are the *support vectors*. The figure below presents an overview of the SVM process.

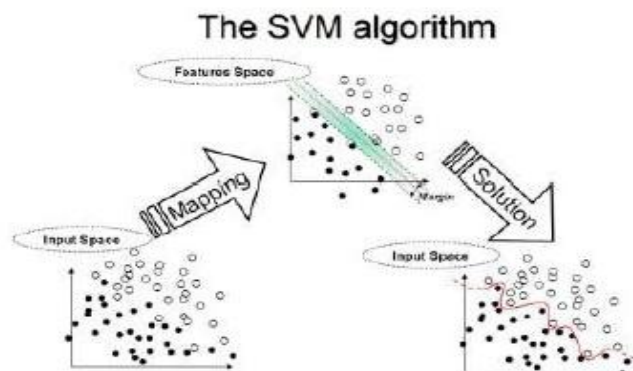


Figure 3.7:Two-Dimensional Example

Before considering N -dimensional hyper planes, let's look at a simple 2-dimensional example. Assume we wish to perform a classification, and our data has a categorical target variable with two categories. Also assume that there are two predictor variables with continuous values. If we plot the data points using the value of one predictor on the X-axis and the other on the Y-axis we might end up with an image such as shown

below. One category of the target variable is represented by rectangles while the other category is represented by ovals [19].

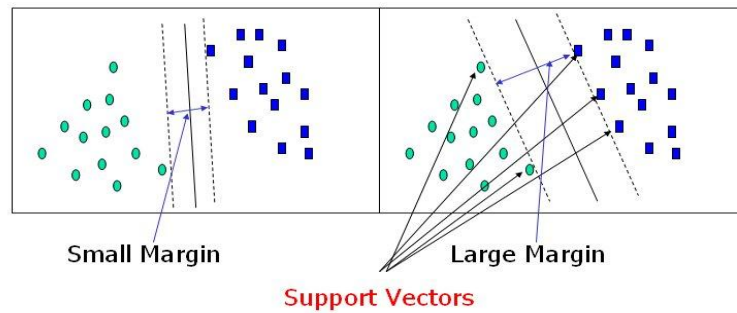


Figure 3.8: Classification of Support Vectors

An SVM analysis finds the line (or, in general, hyper plane) that is oriented so that the margin between the support vectors is maximized. In the figure above, the line in the right panel is superior to the line in the left panel.

CHAPTER 4

PROBLEM STATEMENT

The known NP-hardness results imply that for many combinatorial optimization problems there are no efficient algorithms that find an optimal solution, or even a near optimal solution, on every instance. A heuristic for an NP-hard problem is a polynomial time algorithm that produces optimal or near optimal solutions on some input instances, but may fail on others. The study of heuristics involves both an algorithmic issue (the design of the heuristic algorithm) and a conceptual challenge, namely, how does one evaluate the quality of a heuristic.

We know exact algorithms might need centuries to solve a formidable problem. In such cases heuristic algorithms that find approximate solutions but have acceptable time and space complexity play indispensable role. In present, all known algorithms for NP-complete problems are requiring time that is exponential in the problem size. Heuristics are a way to improve time for determining an exact or approximate solution for NP-problems.

There are many NP-Problems are available in different fields, such as science, engineering and technology. Out of all problems we selected few wellknown problems. We are trying to estimate seven heuristics (such as Hill Climbing, Simulated Annealing, Swarm Intelligence, Tabu Search, Evolutionary Algorithms, Neural Networks and Support vector Machines) for some selected problems. One objective is that, after applying different heuristics for a particular NP-problem, a set of guidelines can be given that how a particular category of heuristics is better for a particular set of problems.

CHAPTER 5

RESULTS AND DISCUSSION

We have estimated seven Heuristic strategies (Hill Climbing, Simulated Annealing, Swarm Intelligence, Tabu Search, Evolutionary Algorithms, Neural Networks and Support vector Machines) for some well-known problems. The corresponding observation for each problem is mentioned below the problem. There is a separate table for each problem that compares different Heuristic Strategies for each problem [22, 23, and 24].

1. Travelling Salesman Problem [14,15,25]

The Travelling Salesman Problem (TSP) is a simple combinatorial problem [12]. It can be stated Very simply:

- A salesman spends his time visiting n cities (or nodes). In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimize the distance travelled?
- There exists an edge between every pair of cities expressing the distance between the two Corresponding cities.
- If there are only 2 cities then the problem is trivial, since only one tour is possible. In either case the number of solutions becomes extremely large for large n , so that an exhaustive search is impractical.

Heuristic Strategy	
Hill Climbing	1. Not effective, Because no agenda is maintained.
Simulated Annealing	1. Current solution wandering from neighbour to neighbour as the computation proceeds. 2. Examines neighbours in random order. 3. Schema leaves several operations and definitions unspecified. 4. As the temperature goes down, the probability of accepting bad moves decreases.
Swarm Intelligence(ACO)	1. Ants leave a trail of pheromones when they explore new areas.

	2. The ant who picked the shortest tour will be leaving a trail of pheromones inversely proportional to the length of the tour. 3. This pheromone trail will be taken in account when an ant is choosing a city to move to, making it more prone to walk the path with the strongest pheromone trail. 4. This process is repeated until a tour being short enough is found.
Tabu Search	1. Implementation of tabu search degrades substantially as N increases. 2. Only makes uphill moves when it is stuck in local optima.
Evolutionary Algorithms	1. without the local optimization. 2. Applying GA to the TSP involves implementing a crossover routine, a measure of fitness, and also a mutation routine.
Neural Networks	1. Multiple random starts were allowed. 2. Best solution they ever found on such an instance was still more than 17% above optimal. 3. very sensitive 4. N^2 neurons are required
Support Vector Machines	1. Not tested

Table 5.1: Shows comparison of different heuristic strategies for Travelling Salesmen Problem.

There are a number of algorithms used to find optimal tours, but none are feasible for large instances since they all grow exponentially. We can get down to polynomial growth if we settle for near optimal tours. We gain speed and speed at the cost of tour quality. So the interesting properties of heuristics for the TSP are mainly speed and closeness to optimal solutions.

If shorter tours are wanted and significantly more time is available, both simulated annealing and genetic algorithms can for many instances find better tours than could be found in the same time by performing multiple independent runs of other heuristics. Within the running time bounds of these algorithms, no tabu search, simulated annealing, genetic, or neural net algorithm has yet been developed that provides comparably good tours.

2. Time Table Problem [26, 27, 28]

The problem is to design and implement an algorithm to create a semester course time table by assigning time-slots and rooms to a given set of courses to be run that semester under given constraints. The constraints include avoiding clashes of time-slots and rooms, assigning appropriate rooms and appropriate no. of slots and contact hours to the courses etc.

Heuristic Strategy	
Hill Climbing	1. Not effective, Because no agenda is maintained. 2. It can easily be verified the search space for this kind of problem is very large 3. The best solution within a reasonable small amount of time depends on neighbourhoods.
Simulated Annealing	1. In our experiments we decided to use a reduction factor of 0.9 and an initial acceptance probability of 0.8 to cool down quite slow 2. Current solution wandering from neighbour to neighbour as the computation proceeds. 3. Examines neighbours in random order. 4. Schema leaves several operations and definitions unspecified. 5. The cooling factor decrease Factor is set to 0.9.
Swarm Intelligence(ACO)	1. Not tested
Tabu Search	1. TS is able to find better solution until the end of the computation 2. Implementation of tabu search degrades substantially as N increases. 3. Only makes uphill moves when it is stuck in local

	optima 4.The best regular tabu list length seems to be approx. 40 elements
Evolutionary Algorithms	1.without the local optimization 2. EAs give lower total penalties compared with man-made schedules. 3. The best individual of a generation will survive and 5 % of the individuals. 4. Every resource list of the individual is subject to mutation with a probability of 0.5 %.
Neural Networks	1. Multiple random starts were allowed. 2. Best solution they ever found on such an instance was still more than 17% above optimal. 3.very sensitive 4. N^2 neurons are required
Support Vector Machines	1.Not tested

Table 5.2: Shows comparison of different heuristic strategies for Time Table Problem.

The implementation effectively optimizes the constructed objective function for the modelled population, under sufficiently heavy constraints. The execution is reasonable for a large problem size, which would not be possible with conventional algorithms. A working algorithm of polynomial time complexity has been implemented, where the conventional solutions have exponential complexity. The operators used on population are performing efficiently. The modelling of the population and a Genetic Algorithm are suited to the problem.

3. Generation Expansion Problem [29, 30]

Generation expansion planning has historically addressed the problem of identifying ideal technology, expansion size, sitting, and timing of construction of new plant capacity in an economic fashion and in a manner that ensures installed capacity adequately meets projected demand growth. Generation expansion planning is a challenging problem due to its large-scale, long term, non-linear, and discrete nature of generation unit size. Generation expansion planning (GEP) is to determine WHAT

generating units should be constructed and WHEN generating units should come on line over a long-term planning horizon.

The criteria are to minimize the total cost and/or maximize the reliability with different type of constraints. The total cost is the sum of capital investment and operation cost. The constraint includes capacity constraints, energy constraints, and operation constraints etc.

Heuristic strategy	
Hill climbing	1. Not tested
Simulated Annealing	1. SA iteratively searches the neighbour by adding some random number with the current solution. The best solution is readily accepted. The worst solution is also accepted by comparing with a random number (0, 1), which avoids trapping in local minima. 2. In each step, the algorithm picks a <i>random</i> move. If it improves the objective function ($\Delta E > 0$), it is accepted. Otherwise, the <i>bad</i> move is only accepted with a probability $e^{\Delta E/T}$
Swarm Intelligence(ACO)	1. Then ants are placed randomly in the first stage and allowed to move based on the probability. After the ants completed the tour, the objective function and fitness function values for the individuals are calculated.
Tabu Search	1. This reduces the size of neighbourhood. Then the combinations between the capacities should be taken as the neighbours keeping other two stages unaltered. Similarly for the other two stages, the neighbours are determined. The candidate list is formed with the combination of these neighbors. The best neighbour among the candidate list is moved to the “ <i>Tabu list</i> ” for a pre-specified number of generations.
Evolutionary Algorithms	1. The uniform binary window and head-to-head crossover have SR of 60%, whereas the arithmetic crossover has 30%.

	2. without the local optimization.
Neural Networks	1. Multiple random starts were allowed. 2. Very sensitive.
Support Vector Machines	1. Not tested

Table 5.3: Shows comparison of different heuristic strategies for Generation Expansion Problem.

The emerging technologies, such as Hill climbing, simulated annealing, genetic algorithm, neural networks, etc are powerful to solve large scale generation expansion planning problems. The advantage of SA approach lies in that it is capable of not only handling a mixed-integer nonlinear programming but also searching toward a global optimal solution. When we are using GA, the more processors the system uses, the less the computation time is required, but the more cost is needed. The number of processors used is trade-off between computation speed and cost.

Therefore we conclude that, Simmulated Annealing approach is the best heuristic for finding the global optimum solution.

4. Vertex Cover Problem [24].

In computer science, the **Vertex Cover Problem** or **Node Cover Problem** is an NP-complete problem and was one of Karp's 21 NP-complete problems. It is often used in complexity theory to prove NP-hardness of more complicated problems.

Definition: A vertex cover for an undirected graph $G = (V, E)$ is a subset S of its vertices such that each edge has at least one endpoint in S . In other words, for each edge ab in E , one of a or b must be an element of S .

Heuristic Strategy	
Hill Climbing	1. Not effective, Because no agenda is maintained.
Simulated Annealing	1. Simulated annealing starts initially with an arbitrary solution and then repeatedly tries to make improvements to it locally. 2. Current solution wandering from neighbour to neighbour as the computation proceeds. 3. Examines neighbours in random order.

	4. A new solution is accepted with a probability that is based on the difference $Q_{old} - Q_{new}$ between the quality of the old and new solutions and on an (artificial) temperature T, which is gradually decreasing throughout the process. 5. Schema leaves several operations and definitions unspecified.
Swarm Intelligence	1. Not tested
Tabu Search	1. Implementation of tabu search degrades substantially as N increases. 2. Only makes uphill moves when it is stuck in local optima.
Evolutionary Algorithms	1. without the local optimization. 2. GA starts by generating a random population of candidate solutions. At each iteration a population of promising solutions is first selected. Variation operators are then applied to this selected population to produce new candidate solutions. Specifically, crossover is applied to exchange partial solutions between pairs of solutions and mutation is used to perturb the resulting solutions. Here we use uniform crossover and bit-flip mutation to produce new solutions. 2. The algorithm stops when the population reaches a stable state.
Neural Networks	1. Multiple random starts were allowed. 2. Best solution they ever found on such an instance was still more than 17% above optimal. 3. very sensitive 4. N^2 neurons are required
Support Vector Machines	1. Not tested

Table 5.4: Shows comparison of different heuristic strategies for Vertex Cover Problem.

Vertex Cover, a well known NP-Complete combinatorial optimization problem having practical applications in fields such as networking and scheduling. After comparing different heuristics for vertex cover problem, we conclude that either SA or GA produce best optimal solution.

5. Mean Flow Time Open Shop Scheduling [21, 22]

The open shop scheduling problem can be described as follows. A set of n jobs $J_1, J_2, \dots, J_n$ has to be processed on a set of m machines $M_1, M_2, \dots, M_m$. The processing of job J_i on machine M_j is denoted as operation (i, j) , and the sequence in which the operations of a job are processed on the machines is arbitrary. All processing times of the operations are assumed to be given in advance. Each machine can process at most one job at a time and each job can be processed on at most one machine at a time.

Heuristic Strategy	
Hill Climbing	1. Not effective, Because no agenda is maintained.
Simulated Annealing	1. Current solution wandering from neighbour to neighbour as the computation proceeds. 2. A non-improving neighbor is accepted with probability $\exp(-\Delta/T)$ 3. Schema leaves several operations and definitions unspecified. 4. As the temperature goes down, the probability of accepting bad moves decreases.
Swarm Intelligence	1. Probabilistic solution construction mechanism using the pheromone model ('pheromone trail parameters') 2. A number of ants probabilistically construct solutions 3. Application of a randomized Beam-Append procedure (for generating nondelay schedules) 4. Pheromone values encode for any two related operations the desirability to perform a particular operation first 5. Use of an iterative procedure to improve the constructed solutions.

Tabu Search	1. Implementation of tabu search degrades substantially as N increases. 2. Only makes uphill moves when it is stuck in local optima. 3. Search moves in each iteration to the best non-tabu neighbour investigated. 4. Tabu restriction is enforced by a neighbourhood-specific tabu list 5. Neighbourhoods: same as for simulated annealing 6. Constant length of the tabu list
Evolutionary Algorithms	1. Without the local optimization. 2. Search technique based on the mechanism of natural selection and genetics 3. Initial population: randomly generated non-delay schedules 4. Chromosome (solution) representation: rank matrix of a sequence graph 5. Genetic operators: (1) Mutation: change the rank of exactly one operation and maintain the relative order of the other operations (2) Crossover: exchange the ranks of a randomly chosen set of operations, maintain the relative order of the remaining operations, and combine both parts to a feasible rank matrix
Neural Networks	1. Multiple random starts were allowed. 2. Best solution they ever found on such an instance was still more than 17% above optimal. 3. Very sensitive 4. N^2 neurons are required
Support Vector Machines	1. Not tested

Table 5.5: Shows comparison of different heuristic strategies for Mean Flow Time Open Shop Scheduling.

For the open shop problem with mean flow time minimization, the choice of an appropriate constructive solution procedure strongly depends on the relationship between the number n of jobs and the number m of machines while the processing times have less influence. Constructive heuristics based on matching procedures do not work well for mean flow time minimization.

For $n < m$

We conclude that in most cases, the genetic algorithm outperforms the other algorithms, but for several problem types simulated annealing is on the first rank. However, the computational times for the genetic algorithms are considerably larger than for simulated annealing (although the same number of solutions has been generated).

For $n = m$

For the majority of problems, the genetic algorithms yield clearly the best results. Both variants, i.e. the initial population filled with random solutions as well as including some solutions obtained by the constructive algorithms contribute best values. The tabu search algorithm is not competitive and yields only marginal improvements of the best constructive solution. Summarizing, the genetic algorithms are clearly the best among all algorithms tested.

For $n > m$

We conclude that the genetic algorithm with an initial population using some of the constructive algorithms gives the best results, followed by simulated annealing.

CHAPTER 6

CONCLUSION

NP-Problems are problems that are not currently solvable in polynomial time. However, they are polynomially equivalent in the sense that any NP-Problem can be transformed into any other in polynomial time.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- **Approximation:** Instead of searching for an optimal solution, search for an "almost" optimal one.
- **Randomization:** Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. See Monte Carlo method.
- **Restriction:** By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- **Parameterization:** Often there are fast algorithms if certain parameters of the input are fixed.
- **Heuristic:** An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result.

Out of all these Heuristic is the technique, which finds solutions among all possible ones, but they do not guarantee that the best to be found; therefore they may be considered as approximate and not accurate algorithms. These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best.

In our thesis we selected different NP-Problems (Travelling Salesmen Problem (TSP), Time Tabling Problem (TTP), Generation Expansion Problem (GEP), Vertex Cover Problem (VCP), Mean Flow Time Open Shop Scheduling Problem) from different fields and we estimated seven Heuristic strategies (such as Hill Climbing, Simmulated Annealing, Swarm Intelligence, Tabu Search, Evolutionary Algorithms, Neural

Networks and Support vector Machines) for the above mentioned problems. We made a set of guidelines can be given that how a particular category of heuristics is better for a particular set of problems. We made an overall conclusion as follows:

Hill-climbing techniques belong to the class of Local search. Such algorithms consistently replace the current solution with the best of its neighbours if it is better than the current. Hill-climbing algorithm is effective, but it has a significant drawback called *premature convergence*. Since it is “greedy”, it always finds the nearest local optima of low quality. The goal of modern heuristics is to overcome this disadvantage.

Simulated annealing algorithm, uses an approach similar to hill-climbing, but occasionally accepts solutions that are worse than the current. The probability of such acceptance is decreasing with time. SA is the best approach for finding the optimal solution for TSP and Minimum Vertex Cover problem.

Tabu search extends the idea to avoid local optima by using memory structures. The problem of simulated annealing is that after “jump” the algorithm can simply repeat its own track. Tabu search prohibits the repetition of moves that have been made recently.

Swarm intelligence is an artificial intelligence technique, based on the study of collective behaviour in decentralized, self-organized, systems. Two of the most successful types of this approach are Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO). In ACO artificial ants build solutions by moving on the problem graph and changing it in such a way that future ants can build better solutions. PSO deals with problems in which a best solution can be represented as a point or surface in an n-dimensional space. The main advantage of swarm intelligence techniques is that they are impressively resistant to the local optima problem.

Any application of GAs involves a selection of an appropriate representation of sample points in the function space, and the creation of a function that describes the behaviour of the space to be searched. Unfortunately, many NP-Complete problems have constrained spaces that do not map well to bit string representations. The TSP is a classic example of such a problem.

Any application of *neural networks* involves selection of an appropriate network representation. Furthermore, a constraint satisfaction approach requires a specification of the domain specific constraints. These constraints must be mapped into an energy

function that adequately describes the space to be searched. In general, these tasks can be difficult. The problem of premature convergence is also critical for Neural Networks.

Support Vector Machines (SVMs) extend the ideas of Neural Networks. They successfully overcome premature convergence since convex objective function is used, therefore, only one optimum exists.

After analysing the different heuristics for some well-known problems, we conclude that, based on the problem characteristics different heuristics are efficient for different problems. We can't say particular heuristic is efficient for all NP-problems. Based on problem criteria and characteristics one of the heuristic is efficient.

For the TSP , If shorter tours are wanted and significantly more time is available, both simulated annealing and genetic algorithms can for many instances find better tours than could be found in the same time by performing multiple independent runs of other heuristics. Within the running time bounds of these algorithms, no tabu search, simulated annealing, genetic, or neural net algorithm has yet been developed that provides comparably good tours.

For Time Table problem, the execution is reasonable for a large problem size, which would not be possible with conventional algorithms. A working algorithm of polynomial time complexity has been implemented, where the conventional solutions have exponential complexity. The operators used on population are performing efficiently. The modelling of the population and a Genetic Algorithm are suited to the problem.

For Generation Expansion Problem, The advantage of SA approach lies in that it is capable of not only handling a mixed-integer nonlinear programming but also searching toward a global optimal solution. When we are using GA, the more processors the system uses, the less the computation time is required, but the more cost is needed. The number of processors used is trade-off between computation speed and cost. Therefore SA approach is suitable for Generation Expansion Problem.

Like this for different problems different heuristics are suitable to find best optimal solution.

ANNEXURE-I

REFERENCES

- [1] S. A. Cook. "An overview of computational complexity", in Communication of the ACM, vol. 26, no. 6, June 1983, pp. 401–408.
- [2] T. Cormen, Ch. Leiserson, R. Rivest. *Introduction to algorithms*. MIT Press, 1989.
- [3] M. R. Garey, D. S. Johnson. *Computers and Intractability*. Freeman&Co, 1979.
- [4] R. B. Boppana and M. Sipser. *The complexity of finite functions*. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 759–804. Elsevier and The MIT Press, 1990.
- [5] Larrabee, T. (1990). *Efficient Generation of Test Patterns Using Boolean Satisfiability*. PhD thesis, Department of Computer Science, Stanford University. Technical Report STAN-CS-90-1302
- [6] A.F. Archer, "A modern treatment of the 15 puzzle," *Amer. Math. Monthly* 106 (1999) 793-799.
- [7] 4. Martello, S., Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations*.Wiley, New York (1990)
- [8] Schofield, P. D. A. *Complete solution of the Eight Puzzle*. In *Machine Intelligence* by N. L. Collins and D. Michie, pages 125-133. American Elsevier, 1967
- [9] Richard Kaye. Minesweeper is NP–Complete. *Mathematical Intelligencer* , volume 22 number 2, pages 9–15, 2000.*Tetris DS* manual. Nintendo, 2006
- [10] Rubin, Frank, "A Search Procedure for Hamilton Paths and Circuits". Journal of the ACM, Volume 21, Issue 4. October 1974. ISSN 0004-5411
- [11] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem*. Wiley, Chichester, 1985.
- [12] M. E. Aydin, T. C. Fogarty. "A Distributed Evolutionary Simulated Annealing Algorithm for Combinatorial Optimization Problems", in *Journal of Heuristics*, vol. 24, no. 10, Mar. 2004, pp. 269–292.
- [13] R. Battiti. "Reactive search: towards self-tuning heuristics", in *Modern heuristic search methods*. Wiley&Sons, 1996, pp. 61-83.
- [14] R. Eberhart, Y. Shi, and J. Kennedy. *Swarm intelligence*. Morgan Kaufmann, 2001.

- [15] J.C. Crput, A. Koukam, T. Lissajoux, A. Caminada. "Automatic Mesh Generation for Mobile Network Dimensioning Using Evolutionary Approach", in IEEE Transactions on Evolutionary Computation, vol. 9, no. 1, Feb. 2005, pp. 18–30.
- [16] B. Rose, P. Smagt. *An introduction to Neural Networks*. University of Amsterdam, Nov. 1996.
- [17] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, 1995.
- [18] S. Gunn. "Support Vector Machines for Classification and Regression", Technical Report, May 1998, http://www.ecs.soton.ac.uk/_srg/publications/pdf/SVM.pdf.
- [19] Pablo Moscato. *NP Optimization Problems, Approximability and Evolutionary Computation: From Practice to Theory*. in Ph.D. Dissertation, Universidade Estadual de Campinas, Brazil. March 2001.
- [20] Bräsel, H.; Herms, A.; Möorig, M.; Tautenhahn, T.; Tusch, T.; Werner, F.: *Heuristic Algorithms for Open Shop Scheduling to Minimize Mean Flow Time*, Part I: Constructive Algorithms, Preprint 30/05, FMA, Otto-von-Guericke-University Magdeburg, 2005.
- [21] Michael Andresen, Heidemarie Bräsel, Marc Möorig, Jan Tusch, Frank Werner, Per Willenius: *Metaheuristic Algorithms for Open Shop Scheduling to Minimize Mean Flow Time*, Part I: Constructive Algorithms, Preprint 30/05, FMA, Otto-von-Guericke-University Magdeburg, 2006.
- [22] Narendhar Maaroju, Deepak Garg "Choosing the Best Heuristic for A NP-Problem" International Journal of Information Technology & Knowledge Management Issue 2 Vol 1 Year 2008 ISSN 0973-4414, pp 537-547.
- [23] Narendhar Maaroju, K V R Kumar, Deepak Garg, "Approximation Algorithms for NP-problems" CIIT International Journal of Software Engineering and Technology, Issue May-2009 ISSN 0974 – 9748 & Online: ISSN 0974 – 9632.
- [24] K V R Kumar, Narendhar Maaroju, Deepak Garg, "Efficient algorithm for Vertex Cover Problem" CIIT International Journal of Software Engineering and Technology, Issue May-2009 ISSN 0974 – 9748 & Online: ISSN 0974 – 9632.
- [25] D.S. Johnson and L.A. McGeoch, "The Traveling Salesman Problem: A Case Study in Local Optimization", November 20, 1995.
- [26] G. M. White and B. S. Xie, "Examination timetables and tabu search with longer term memory," *Proc. of the International Conference on the Practice and Theory of Timetabling*, 184-201 (2000).

- [27] M. L. Ng, et al, "Solving constrained staff workload scheduling problems using simulated annealing technique," *Proc. of the International Conference on the Practice and Theory of Timetabling*, 355-371 (2000).
- [28] M. Gröbner and P. Wilke. *A general view on timetabling problems*. In Edmund Burke and Patrick De Causmaecker, editors, *Proc. of the 4th International Conference on the Practice and Theory of Automated Timetabling*, pages pp. 221–227. PATAT 2002, August 2002.
- [29] Y.-M. Park, J.-R. Won, J.-B. Park and D.-G. Kim, "Generation expansion planning based on an advanced evolutionary programming," *IEEE Trans. Power Syst.*, vol. 14, no. 1, pp. 299–305, Feb. 1999.
- [30] J. Zhu and M.-Y. Chow, "A review of emerging techniques on generation expansion planning," *IEEE Trans. Power Syst.*, vol. 12, no. 4, pp. 1722–1728, Nov. 1997.

ANNEXURE-II

LIST OF PUBLICATIONS

- [1] Narendhar Maaroju, Deepak Garg “*Choosing the Best Heuristic for A NP-Problem*” International Journal of Information Technology & Knowledge Management Issue 2 Vol 1 Year 2008 ISSN 0973-4414.pp 537-547.