# ON THE COMPUTATIONAL COMPLEXITY OF COMPUTER ALGORITHMS

THESIS

SUBMITTED FOR THE DEGREE
OF
DOCTOR OF PHILOSOPHY IN SCIENCE
(Pure Mathematics)



PRESENTED BY

*ELHAM MOHAMED EL-SHERIF*



Supervised By

**Prof. Dr. Bayoumi I. Bayoumi**
Department of Mathematics
Faculty of Science, Ain Shams University

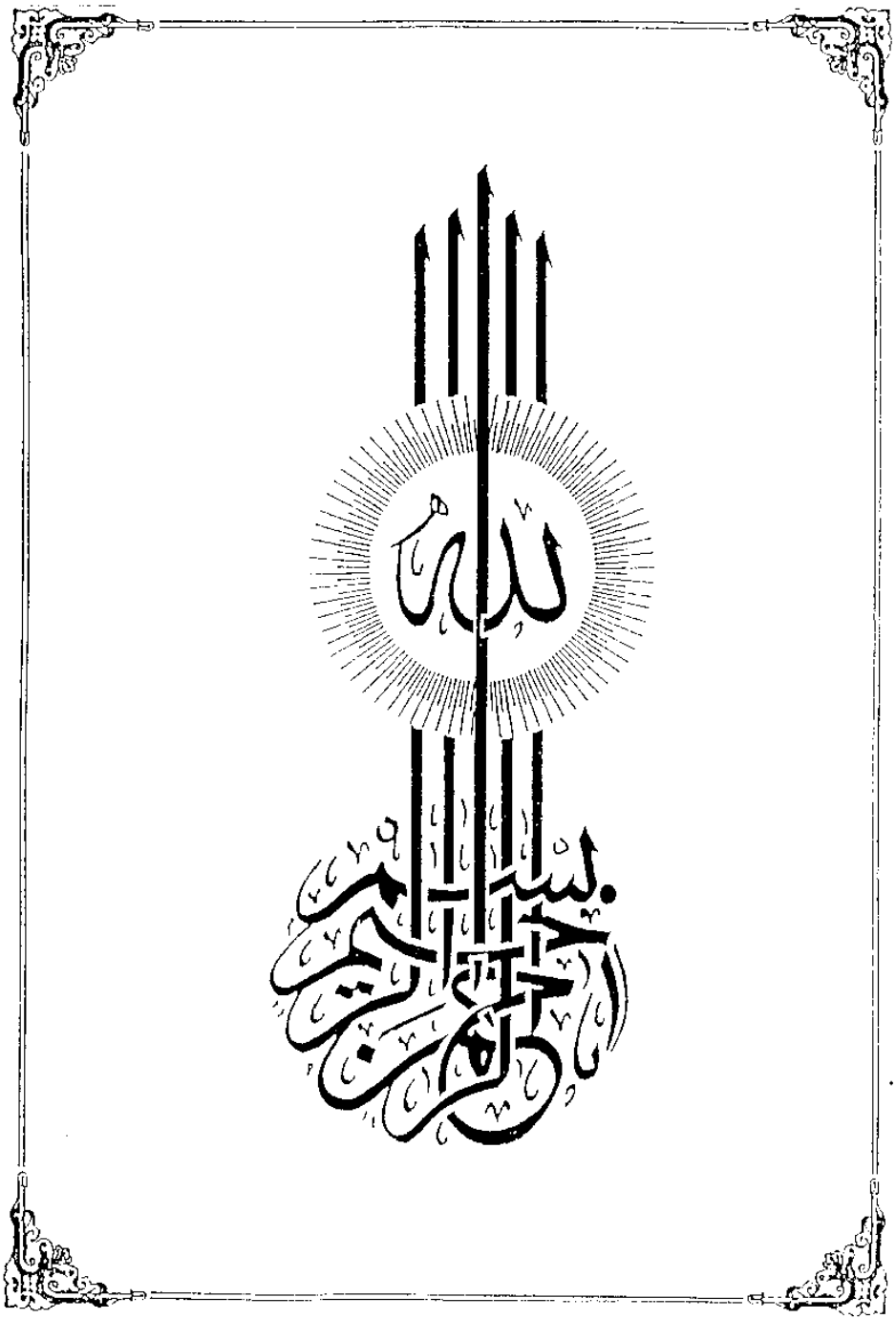
**Dr. Mohamed M. Kouta**
Department of Computer Science and Operations Research
Military Technical College

SUBMITTED TO

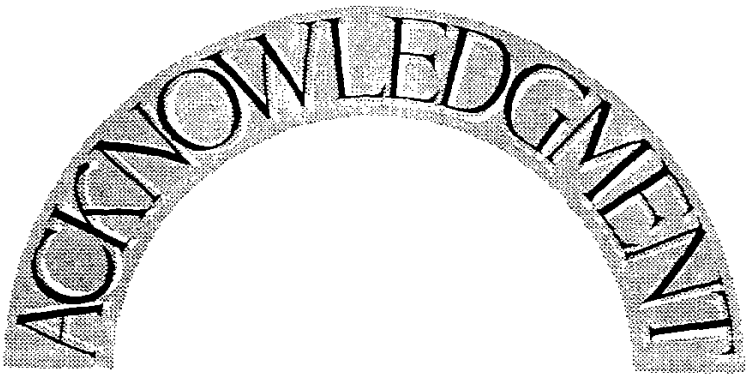
Department of Mathematics
Faculty of Science
Ain Shams University

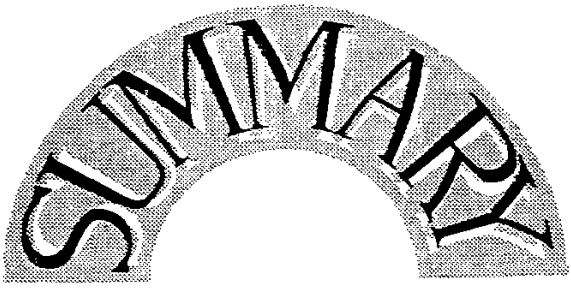
CAIRO, EGYPT

1995



# CONTENTS





The thesis consists of four chapters.

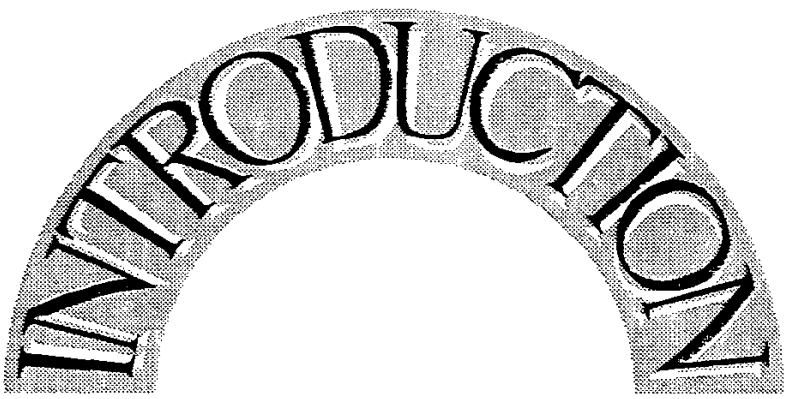
In chapter 1, an introduction to the notion of time complexity of algorithms and the theory of NP-completeness are introduced.

In chapter 2, the scheduling problems and their classification are considered. The problem of scheduling unit execution time tasks on a number of processors under some precedence constraints has been studied. What has been accomplished to date in solving this problem is also presented.

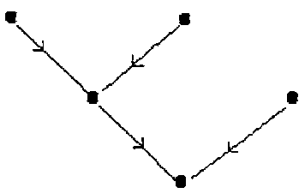
In chapter 3, approximation algorithms and the performance guarantee approach are introduced. This approach is illustrated by considering some algorithms for some known NP-hard problems.

In chapter 4, a calculation of an upper bound on the number of solutions of the m-machine scheduling for N-free posets for unbounded m is given. We also present a polynomial time algorithm to find an optimum schedule of the UET problem on m, unbounded, processors, when the precedence relations between tasks can be represented by a CBC dag with bounded number of blocks.

The results of this chapter have been published in the proceeding of the Fifth Conference on Operation Research and its Military applications 1993, [15].

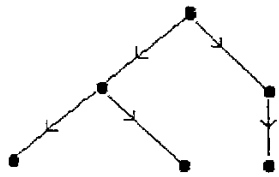


A graph is *connected* if for every pair (x, y) of distinct vertices there is a path from x to y. A graph without any cycles is a *forest*. A *tree* is a connected forest. A directed acyclic graph (dag) is called an *in-tree* if every node lies on a unique directed path from that node to the root see Figure 1-a. A dag is called an *out-tree* if every node lies on a unique path from the root to that node see Figure 1-b.



In-tree

Fig. 1-a



Out-tree

Fig. 1-b

The source in a tree is the root. It follows from the above definition that, in every tree, the root's successor set consists of all other nodes.

A dag is called a *level graph* if its node set can be partitioned into subsets $L_1, L_2, \ldots, L_h$ such that for all $2 \le i \le h$ there is an arc from every node in $L_i$ to every node in $L_{i-1}$ see Figure 2.

A dag is *transitively closed* if $(u, v), (v, w) \in E$ implies that $(u, w) \in E$, and is *transitively reduced* if $(u, w) \in E$ implies that there is no $v \in E$ with $(u, v)$ and $(v, w) \in E$. Edges violating this condition are called transitive or redundant.

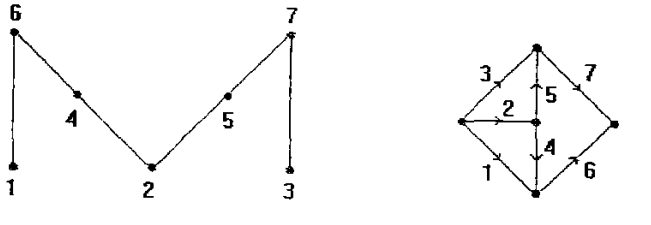
the incomparability graph must have a transitive orientation, that is, an the assignment of direction to its edges such that the resulting digraph is a partial order.

If $P = (V, <)$ is a poset, then $G(P) = (V, E)$ with $E = \{(u, v) | u \sim v\}$ denotes the comparability graph of P. So two vertices are connected by an edge in G(P) iff they are comparable in P.

With the poset $P = (V, <)$ we can associate a directed graph G whose vertices are the elements of the set V. There is a directed arc from vertex u to vertex v if and only if u is an immediate predecessor of v. Since, every poset P may be interpreted as a dag with vertex set V and edge set <, the terms "transitively closed" and "transitively reduced" carry over to partially ordered sets.

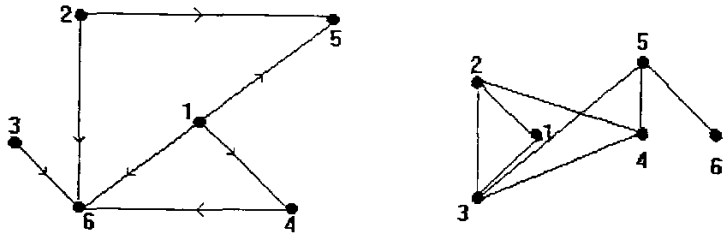
Conversely, a directed acyclic graph induces a partial order in the following way: Let V be the set of vertices of G, then $u < v \Leftrightarrow$ G contains a directed path from u to v, defines a partial order $P = (V, <)$ on V.

If a poset P can be represented by a dag G with exactly one source and one sink, in which the elements of P are represented by edges of G and $a < b$ means that there is a directed path from a to b in G, then P is called *edge-induced* and its dag representation G is called *edge-diagram* or arc-diagram, see Figure 3.




p edge diagram

Fig. 3




(a) An interval order

(b) Incomparability graph of (a)

Fig. 4

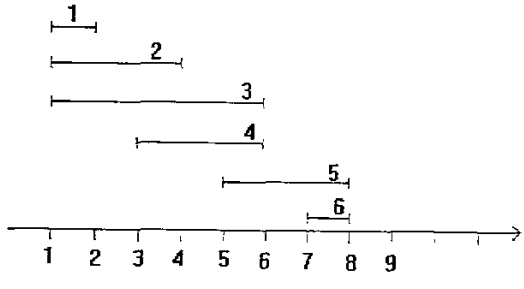



Fig. 5

**Lemma.**

A partial order is an interval order iff its incomparability graph is chordal.

**Proof.**

An interval order is the complement of an interval graph, and hence certainly of a chordal graph (see [24]). □

Define the class of vertex series-parallel (VSP) dags in terms of the subclass of its minimal members. The dags in this subclass are called *minimal vertex series-parallel* (MVSP) and are defined recursively as follows:

The thesis consists of four chapters. The first chapter presents an introduction to some of the notions of computational complexity and discusses the significance of NP-completeness. The definitions involve certain theoretical concepts, such as "languages" and "Turing machines", relating them to the notions of problems. It shows the method for proving that a problem is NP-complete.

Among the NP-complete problems, we choose to study the scheduling problem. A type of scheduling problems called UET problem is introduced in Chapter 2 and will be studied in the remaining of the thesis.

For number of processors greater than 2 and with general dags, the UET problem is NP-complete. No efficient optimization algorithm has been found and there exist simple heuristic algorithms that find feasible schedules quickly. In this case we use an approach to analyzing how good, relative to optimal schedules, are the schedules constructed by such heuristics. This approach deals with proving "performance guarantees" and is discussed in Chapter 3.

In Chapter 4, we solve the UET problem on a general number of processors when the precedence constraints can be represented by a CBC dag with bounded number of blocks. We also find an upper bound on the total number of solutions. The suggested optimization algorithm is presented with an illustrative example.

# CHAPTER I

# THEORY of NP-COMPLETENESS

This chapter covers the basic theory of NP-completeness. Section 1, presents the notion of problems, algorithms and complexity. In section 2, decision problems and their representation as languages will be discussed. In section 3, the one-tape Turing machine is introduced (as our basic model for computation) and is used to define "the class P" of all languages recognizable deterministically in polynomial time. Also, equating "solving" a decision problem with "recognizing" the corresponding language is discussed. In section 4, the one-tape Turing machine is then augmented with a hypothetical guessing ability and is used to define the second important class of decision problems, "the class NP". The relationship between the two classes P and NP is discussed in section 5. In section 6, we introduce the third and most important class, the class of NP-complete problems consisting of the "hardest" problems in NP.

## §1.1 Problems, Algorithms and Complexity

Let us begin with the notion of a problem. For our purposes, a problem will be a general question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified. A problem is described by giving:

1- a general description of all its parameters,

2- a statement of what properties of the answer, or solution, to be satisfied.

An instance of a problem is obtained by specifying particular values for all the problem parameters. Algorithms are general step-

instance. The time complexity function for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size [8].

Let T(n) be the running time of a program on inputs of size n. The units of T(n) will be left unspecified, but we can think of T(n) as being the number of instructions executed on an idealized computer. We say that T(n) is O(f(n)) if there are $c \in \mathbf{R}^+$ and $n_o \in \mathbf{N}$, such that $T(n) \leq cf(n)$ whenever $n \geq n_o$. An algorithm whose running time is O(f(n)) is said to have growth rate f(n) and complexity O(f(n)). To specify a lower bound on the growth rate of T(n) we can use the notation $\Omega(g(n))$, read "big omega of g(n)" or just omega of g(n), to mean that there exists $n_o \in \mathbb{N}$, and $c \in \mathbf{R}^+$ such that $T(n) \geq cg(n)$ $\forall\, n \geq n_o$. It will be assumed that algorithms can be evaluated by comparing their running-time functions, with constants of proportionality neglected [28].

A polynomial time algorithm is defined to be one whose time complexity function is O(p(n)) for some polynomial function p. Any algorithm whose time complexity function cannot be so bounded is called an exponential time algorithm (although it should be noted that this definition includes certain non-polynomial time complexity functions, like $n^{\log n}$, which are not normally regarded as exponential functions) [8].

## §1.2 Decision Problems, Languages.

As a matter of convenience, the theory of NP-completeness is designed to be applied only to decision problems. Such problems have only two possible solutions, either the answer "yes" or the answer "no". Abstractly, a decision problem $\Pi$ consists of a set of $D_\pi$ of instances and a subset $Y_\pi \subseteq D_\pi$ of yes-instances. The standard format used for specifying problems consists of two parts, the first part specifying a *generic instance* of the problem in terms

those that are not encodings of instances of $\Pi$, those that encode instances of $\Pi$ for which the answer is "no" and those that encode instances of $\Pi$ for which the answer is "yes". This third class of strings is the language we associate with $\Pi$ and e, setting

$L[\Pi, e] = \{x \in \Sigma^* : \Sigma$ is the alphabet used by e, and x is the encoding under e of an instance $I \in Y_\pi\}$.

Suppose for example that we are dealing with a problem in which each instance is a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, each edge being an unordered pair of vertices. Such an instance might be described by simply listing all the vertices and edges, by listing rows of the adjacency matrix for the graph, or by listing for each vertex all other vertices sharing a common edge with it (a "neighbour" list). Each of these encodings can give a different input length for the same graph. The input lengths they determine differ at most polynomially from one another, so that any algorithm having polynomial time complexity under one of these encoding schemes also will have polynomial time complexity under all the others. In fact, the standard encoding schemes used in practice for any particular problem always seem to differ at most polynomially from one another. It would be difficult to imagine a "reasonable" scheme for a problem that differs more than polynomially from the standard ones. Although what it means here by reasonable cannot be formalized, the following two conditions capture much of the notion:

1- the encoding of an instance I should be concise and not "paded" with unnecessary information or symbols,

2- numbers occurring in I should be represented in binary (decimal, octal, or in any fixed base other than 1).

The intent of conciseness is that instances of a problem should be described with the brevity one would use in specifying those instances for a computer, without any padding of the input.