Chapter 9

Computational Complexity and Interactability: An Introduction to the Theory of NP

Preview

- Overview
- Lecture Notes
  - Interactability
  - Input Size Revisited
  - The Three General Problem Categories
    - Problems for Which Polynomial-Time Algorithms Have Been Found
    - Problems That Have Been Proven to Be Intractable
    - Problems That Have Not Been Proven to Be Intractable but for Which Polynomial-Time Algorithms Have Never Been Found
- Quick Check
  - The Theory of NP
    - The Sets P and NP
    - NP-Complete Problems
    - The State of NP
    - Complementary Problems
    - NP-Hard, NP-Easy, and NP-Equivalent Problems
  - Handling NP-Hard Problems
    - An Approximation Algorithm for the Traveling Salesperson Problem
    - An Approximation Algorithm for the Bin-Packing Problem
- Quick Check
- Classroom Discussion

- Homework
- Keywords
- Important Links

Overview

The Traveling Salesperson problem and thousands of other problems are equally hard in the sense that if we had an efficient algorithm for any one of them, we would have efficient algorithms for all of them. Such an algorithm has never been found, but it's never been proven that one is not possible. This kind of problem is called **NP-complete** problems, and they are the focus of this chapter. A problem for which an efficient algorithm is not possible is said to be **intractable**. When we are concerned with determining whether or not a problem is intractable, we must be careful about what we call the **input size** in the algorithm. This chapter will cover three general categories into which problems can be grouped as far as intractability is concerned. Finally, we will discuss the theory of *NP* and *NP*-complete problems.

Lecture Notes

Interactability

A problem in computer science is intractable if a computer has difficulty solving it that is it is impossible to solve it with a polynomial-time algorithm. Polynomial-time algorithm is one whose worst-case time complexity is bounded above by a polynomial function of its input size. We stress that *intractability is a property of a problem*; it is not a property of any one algorithm for that problem. For a problem to be intractable there must be no polynomial-time algorithm that solves it. Obtaining a nonpolynomial-time algorithm for a problem does not make it intractable.

Polynomial-time algorithms are usually better than algorithms that are not polynomial-time.

Many algorithms whose worst-case time complexities are not polynomials have efficient running times for many actual instances. This is the case for many backtracking and branch-and-bound algorithms.

There are three general categories of problems that have intractability issues:

1. Problems for which polynomial-time algorithms have been found
2. Problems that have been proven to be intractable
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found

When we are determining whether an algorithm is polynomial-time, it is necessary to be careful about what we call input size.

Input Size Revisited

For a given algorithm, the input size is defined as the number of characters it takes to write the input. To count the characters it takes to write the input, we need to know how the input is encoded. For example, if we encode it in binary, which is used inside computers, the characters used for the encoding are binary digits, and the number of characters it takes to encode a positive integer x is $[\log x] + 1$. The input size is the count of the number of bits it takes to encode them.

If the largest integer is L, and we encode each integer in the number of bits needed to encode the largest, then it takes about $\lg L$ bits to encode each of them. The input size for the n integers is therefore about $n \lg L$.

But instead if we choose to encode the integers in base 10, then the characters used for the encoding are decimal digits, it takes about $\log L$ characters to encode the largest, and the input size for the n integers is about $n \log L$. Because $n \log L = (\log 2)(n \lg L)$, an algorithm is polynomial-time in terms of one of these input sizes if and only if it is polynomial-time in terms of the other.

If we restrict ourselves to "reasonable" encoding schemes, than the particular encoding scheme used does not affect the determination of whether an algorithm is polynomial-time. There does not seem to be a satisfactory formal definition of "reasonable." For any algorithm in this text, any base other than 1 could be used to encode an integer without affecting whether or not the algorithm is polynomial-time. We could, therefore, consider any such encoding system to be reasonable. Encoding in base 1, which is called unary form, would not be considered reasonable.

The **worst-case time complexity** of the algorithm is defined as the maximum number of steps done by the algorithm for an input size of *s*. *W* (*s*) is called the worst-case time complexity of the algorithm.

A **step** can be considered the equivalent of one machine comparison or assignment or one bit comparison or assignment. Each step constitutes one execution of the basic operation. We used *s* instead of *n* for the input size because:

1. The parameter *n* to our algorithms is not always a measure of input size.
2. When *n* is the measure of the input, it is not ordinarily a precise measure of it.

According to this definition, we must count all the steps done by the algorithm. Using Exchange Sort, we assume that the keys are positive integers and that there are no other fields in the records. If the integers are sufficiently large, they cannot be compared or assigned in one step by the computer. Therefore, we should not consider one key comparison or one assignment as one step. To keep our analysis machine-independent, we consider one step to be either one bit comparison or one bit assignment.

We can obtain similar results for all algorithms, which we have shown to be nonpolynomial-time, remain nonpolynomial-time when we are precise about the input size. We see that when *n* is a measure of the amount of data in the input, we obtain correct results concerning whether an algorithm is polynomial-time by simply using *n* as the input size.

In algorithms such as the prime-checking algorithm, we call *n* a **magnitude** in the input. We've sheen other algorithms whose time complexities are polynomials in terms of magnitudes but are not polynomials in terms of size.

Al algorithm whose worst-case time complexity is bounded above by a polynomial function of its size and magnitudes is called **pseudopolynomial-time**. Such an algorithm can often be quite useful because it is inefficient only when confronted with instances containing extremely large numbers.

The Three General Problem Categories

We have three general categories in which problems can be grouped as far as intractability is concerned.

Problems for Which Polynomial-Time Algorithms Have Been Found

Any problem for which we have found a polynomial-time algorithm falls into this category. Because *n* is a measure of the amounts of data in the inputs to these algorithms, they are all polynomial-time. The list of these types of algorithms is endless. There are algorithms that are not polynomial-time for many of these problems. This is the case with the Chained Matrix Multiplication algorithm. Other problems for which we have developed polynomial-time algorithms but for which the brute-force algorithms are nonpolynomial include the Shortest Paths problem, the Optimal Binary Search Tree problem and the Minimum Spanning Tree problem.

Problems That Have Been Proven to Be Intractable

There are two types of problems in this category. The first type is problems that require a nonpolynomial amount of output such as the problem of determining all Hamiltonian Circuits. If there was an edge from every vertex to every other vertex, there would be (n - 1)! such circuits. To solve the problem, an algorithm would have to output all of these circuits, which means that our request is not reasonable.

Remember in Chapter 5 we were stating the problems so as to ask for all solutions because we could then present less-cluttered algorithms, but that each algorithm could easily be modified to solve the problem that asks for only one solution. Although it is important to recognize that this type of intractability problem ordinarily pose no difficulty. It is usually straightforward to recognize that a nonpolynomial amount of output is being requested, and once we recognize this, we realize that we are asking for more information than we can possibly use.

The second type of intractability occurs when our results are not reasonable, and we can prove that a problem cannot be solved in polynomial time. There are relatively few such problems. The first ones were *undecidable* problems. They are undecidable because it has been proven that algorithms that can solve them cannot exist. The most well-known of these problems is the Halting problem.

All problems that to this date have been proven intractable have also been proven not be in the set *NP.* Most problems that appear to be intractable are in the set *NP*.

Problems That Have Not Been Proven to Be Intractable but for Which Polynomial-Time Algorithms Have Never Been Found

This category includes any problem for which a polynomial-time algorithm has never been found, but yet no one has ever proven that such an algorithm is not possible. Many problems belong in the category such as 0-1 Knapsack, the Sum-of Subsets, Traveling Salesperson, m-Coloring, Hamiltonian Circuits, and the problem of abductive interference in a Bayesian network.

We have found branch-and-bound algorithms, backtracking algorithms, and other algorithms for these problems that are efficient in many instances.

Quick Check

1. An algorithm whose worst-case time complexity is bounded above by a polynomial function of its size is called a(n) ____.
   Answer: Polynomial- time algorithm
2. The input size is defined as the number of characters it takes to *read* the input. (True or False)
   Answer: False
3. Chained Matrix Multiplication algorithm is an example of problem for which polynomial time algorithms have been found. (True or False)
   Answer: True
4. In the Problems that have been proven to be intractable, there exists ____ types.
   Answer: two

The Theory of NP

It will be more convenient to develop this theory restricting ourselves to decision problems. The output of a decision problem is a simple “yes” or “no” answer. When we introduced some of these problems previously, we presented them as optimization problems, which means that the output is an optimal solution. Each optimization problem has a corresponding decision problem.

For the Traveling Salesperson problem, the 0–1 Knapsack problem, the Graph–Coloring problem and the Clique problem, we have not found polynomial-time algorithms for either the decision problems or optimization problems. However, if we could find a polynomial-time algorithm for the optimization problem, we would also have a polynomial-time algorithm for the corresponding decision problem. This is so because *a solution to an optimization problem produces a solution to the corresponding decision problem*.

The Sets *P* and *NP*

P is the set of all decision problems that can be solved by polynomial-time algorithms. All decision problems for which we have found poly-time algorithms are certainly in *P*. For example, the problem of determining whether a key is present in an array, the problem of determining whether a key is present in a sorted array, and the decision problems corresponding to the optimization problems are all in *P*.

Can a decision problem for which we have not found a polynomial-time algorithm also be in P? An example is the Traveling Salesperson problem. Even though no one has ever created a polynomial-time algorithm that solves this problem, no one has ever proven that it cannot be solved with a polynomial time algorithm. Therefore, it could *possibly* be in *P*. To know that a decision problem is not in *P*, we have to *prove* that it is not possible to develop a polynomial-time algorithm for it.

What decision problems are not in *P*? We do not know whether the decision problems in examples 9.2 – 9.5 are in *P*. There are thousands of decision problems in this category. There are relatively few decision problems which we *know* are not in *P*. These are decision problems for which we have proven that polynomial-time algorithms are not possible.

Next we define a possibly broader set of decision problems that includes the problems in 9.2 –9.5. To motivate this definition let's examine the Traveling Salesperson problem again. Suppose someone claimed to know that the answer to some instance of this problem is "yes." It would be reasonable for us to ask the person to "prove" this claim by actually producing a tour with a total weight no greater than *d.* If the person produced something, we could write this algorithm to verify whether what the person produced was a tour with weight no greater than *d.*

[See algorithm on page 407]

This algorithm first checks to see if *S* is indeed a tour. If it is, the algorithm then adds the weights on the tour. If the sum of the weights is no greater than *d*, it returns "true."

To state the notion of polynomial-time verifiability more concretely, we introduce the concept of a **nondeterministic algorithm**. This type of algorithm is composed of two separate stages:

1. **Guessing (nondeterministic) Stage**: Given an instance of a problem, this stage simply produces some string *S*. The string can be thought of as a guess at the solution.

2. **Verification (deterministic) Stage**: The instance and the string *S* are the input to this stage. This stage then proceeds in an ordinary deterministic manner either (1) halting with an output of "true," (2) halting with an output of "false," or (3) not halting at all.

The guessing state is nondeterministic because unique, step-by-step instructions are not specified for it. The machine is allowed to produce any string in an arbitrary manner.

We can say a nondeterministic algorithm "solves" a decision problem if:

1. For any instance in which the answer is "yes," there is some string *S* for which the verification stage returns "true."

2. For any instance for which the answer is "no," there is no string for which the verification stage returns "true."

NP is the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms. The *NP* stands for nondeterministic polynomial. For a decision problem to be *NP*, there must be an algorithm that does the verification in polynomial time. Because this is the case for the Traveling Salesperson Decision problem, that problem is in *NP*. It must be stressed that this one does not mean that we have a polynomial-time algorithm that solves the problem. If the answer for a particular instance

of that problem were "yes," we might try all tours in the nondeterministic stage before trying one for which *verify* returns "true." If the answer for an instance were "no," solving the problem using this technique would require that all the tours be tried.

There are thousands of other problems that no one has been able to solve in polynomial-time algorithms but have proven to be in *NP* because polynomial-time nondeterministic algorithms have been developed for them. There is also a large number of problems that are trivially in *NP* because *every problem in P is also in NP.* The only problems that have been proven not to be in *NP* are the same problems that have been proven to be intractable.

To show that $P \neq NP$, we would have to find a problem in *NP* that is not in *P,* whereas to show that $P = NP$, we would have to find a polynomial-time algorithm for each problem in *NP*. Many researchers doubt that *P* equals *NP*.

NP-Complete Problems

The problems in Examples 9.2 – 9.5 may not appear to have the same difficulty. Our dynamic programming algorithm (Algorithm 3.11) for the Traveling Salesperson problem is worst-case. On the other hand, the dynamic programming algorithm in Section 4.4 for the 0–1 Knapsack problem is worst-case. The state space tree in the branch-and-bound algorithm (Algorithm 6.3) for the Traveling Salesperson problem has $(n - 1)!$ leaves, whereas the one in the branch-and-bound algorithm (Algorithm 6.2) for the 0 – 1 Knapsack problem is Y (nW), which means that it is efficient as long as the capacity W of the sack is not extremely large. It seems that the 0 – 1 Knapsack problem is inherently easier than the Traveling Salesperson problem. We show that these two problems, the examples in 9.2 – 9.5, and thousands of other problems are all equivalent in that if one is in *P*, they must all be in *P.* These problems are called ***NP* -complete**.

A **logical (Boolean) variable** is a variable that can have one of two values: true or false. A **literal** is a logical variable or the negation of a logical variable. A **clause** is a sequence of literals separated by the logical or the operator (V). A logical expression in **conjunctive normal form (CNF)** is a sequence of clauses separated by the logical **and** operator (^).

It is easy to write a polynomial-time algorithm that takes as input a logical expression in CNF and a set of truth assignments to the variables and verifies whether the expression is true for that assignment. Therefore the problem is in *NP*. No one has ever found a polynomial-time algorithm for this problem, and no one has ever proven that it cannot be solved in polynomial time. So we do not know if it is in *P*. Cook (1971) published a paper proving that if CNF-Satisfiability is in *P*, then $P = NP$.

Suppose we want to solve decision problem A, and we have an algorithm that solves decision problem B. Suppose further that we can write an algorithm that creates an instance *y* of Problem B from every instance *x* of Problem A such that an algorithm for Problem B answers "yes" for *x*. Such an algorithm is called a **transformation algorithm.** Refer to the text for a more detailed discussion.

The State of *NP*

Figure 9.3 shows *P* as a proper subset of *NP*, but it may not be the same set. Presburger, Arithmetic, the Halting problem, and many other decision problems that are not in *NP* are not *NP*-complete. A decision problem that is in *NP* and is not *NP*-complete is the trivial decision problem that answers "yes" or "no" for all instances. No one has been able to prove that there is a problem in *NP* that is neither in *P* nor *NP*-complete. It has been proved that if $P \neq NP$, such a problem must exist. Refer to the text for a more detailed discussion of Theorem 9.7.

Complementary Problems

In general, the **complementary problem** to a decision problem is the problem that answers "yes" whenever the original problem answers "no" and vice versa. If we found an ordinary deterministic polynomial-time algorithm for a problem, we could have a deterministic polynomial-time algorithm for its complementary problem. However, finding a polynomial-time nondeterministic algorithm for a problem does not automatically produce a polynomial-time nondeterministic algorithm for its complementary. Indeed, no one has ever shown that the complementary problem to any known *NP*-complete problem is in *NP*. On the other hand, no one has ever proven that some problem is in *NP* whereas its complementary problem is not in *NP.* Refer to the text for a more detailed discussion.

NP-Hard, NP-Easy, and NP-Equivalent Problems

A problem *B* is called ***NP-hard*** if, for some *NP*-complete problem A, A (T B. Clearly, every *NP*-complete problem is *NP*-hard. *The optimization problem corresponding to any NP-complete problem is NP-hard.* Refer to the text for an example that uses the definition of Turing reducibility to show this for the Traveling Salesperson problem.

What problems are not *NP*-hard? We do not know if there is any such problem. Indeed, if we were to prove that some problem was not *NP*-hard, we would be proving that *P* = *NP*. The reason is that we would then have an actual rather than a hypothetical polynomial-time algorithm for some *NP*-hard problem. Therefore, we could solve each problem in *NP* in polynomial-time using the Turing reduction from the problem to the *NP*-hard problem.

If a problem is *NP*-hard, it is at least as hard as the *NP*-complete problems. A problem *A* is called ***NP-easy*** if, for some problem *B* in *NP*, A (T B. if P = NP, then a polynomial-time algorithm exists for all *NP*-easy problems. What problems are *NP*-easy? Obviously, the problems in *P*, the problems in *NP*, and nondecision problems for which we have found polynomial-time algorithms are all N*P*-easy. The optimization problem, corresponding to an *NP*-complete decision problem, can usually be shown to be *NP*-easy.

A problem is called ***NP-equivalent*** if it is both *NP*-hard and *NP*-easy.

Handling *NP*-Hard Problems

In the absence of polynomial-time algorithms for problems known to be *NP*-hard, what can we do about solving such problems? One way is the backtracking and branch-and-bound algorithms, which are all worst-case nonpolynomial-time. However, they are often efficient for many large instances.

Another approach is to find an algorithm that is efficient for a subclass of instances of an *NP*-hard problem. A third approach is to develop **approximation algorithms**. An approximation algorithm for an *NP*-hard optimization problem is an algorithm that is not guaranteed to give optimal solutions, but rather yields solutions that are reasonably close to optimal.

An Approximation Algorithm for the Traveling Salesperson Problem

If a weighted, undirected graph $G = (V, E)$ be given such that

1. There is an edge connecting every two distinct vertices.
2. If $W(u, v)$ denotes the weight on the edge connecting vertex u to vertex v, then, for every other vertex y,

$$W(u, v) \leq W(u, y) + W(y, v)$$

The second condition, called the **triangular inequality** is shown in Figure 9.10. It is satisfied if the weights represent actual distances between cities. The first condition implies that there is a two-way road connecting every city to every other city. The problem is to find the shortest path (optimal tour) starting and ending at the same vertex and visiting each other vertex exactly once. It can be shown that this variant of the Traveling Salesperson is also *NP* -hard.

Notice that the graph in this variant of the problem is undirected. If we remove any edge from an optimal tour for such a graph, we have a spanning tree for the graph. Therefore the total weight of a minimum spanning tree must be less than the total weight of an optimal tour. By going twice around the spanning tree, we can convert it to a path that visits every city. The resulting path may visit some vertices more than once. We can convert the path to one that does not do this by taking shortcuts. Refer to Figure 9.11 for an approximation to an optimal tour from a minimum spanning tree. Refer to the text for a more detailed discussion of Theorem 9.6.

An Approximation Algorithm for the Bin-Packing Problem

Let n items with sizes *s*1, *s*2,..., s*n*, where 0 < si <= 1, be given, and suppose we are to pack the items in bins, where each bin has a capacity of 1. The problem is to determine the minimum number of bins necessary to pack all the items.

This problem has been shown to be *NP*-hard. A very simple approximation algorithm for this problem is called **first-fit**. This strategy places an item in the first bin in which it fits. If it does not fit, a new bin is started. It is also advisable to pack the items in nonincreasing order. This strategy is called **nonincreasing first fit**. Refer to the text for a more detailed discussion of the algorithm.

One way to gain further insight into the quality of an approximation algorithm is to run empirical tests comparing the solutions obtained from the approximation with the optimal solutions. Our approximation algorithm for the Bin-Packing problem has been extensively tested for large values of *n*. In the case of the Bin-Packing problem, we do not need to actually compute optimal solutions to gain insight into the quality of the approximations. Instead, we compute the amount of unused space in the bins used by the approximation algorithm. The number of extra bins used by that algorithm can be no more than the amount of unused space.

Quick Check

1. The set *P* is the set of all decision problems that can not be solved by polynomial-time algorithms (True or False)
   Answer: False
2. A nondeterministic algorithm whose verification stage is a polynomial-time algorithm is called, ____.
   Answer: polynomial-time nondeterministic algorithm
3. A problem is called *NP*-equivalent if it is both ____ and ____.
   Answer: *NP*-hard, *NP*-easy
4. A(n) ____ for an NP-hard optimization problem is an algorithm that is not guaranteed to give optimal solutions, but rather yields solutions that are reasonably close to optimal.
   Answer: approximation algorithm

Classroom Discussion

- Discuss three problems that have polynomial time algorithms
- Discuss a problem and two encoding schemes for its input.

Homework

Assign Exercises 4, 17, and 26.

Keywords

- **0–1Knapsack Decision problem –** determine, for a given profit *P*, whether it is possible to load the knapsack so as to keep the total weight no greater than *W.*
- **0–1Knapsack Optimization problem –** determine the maximum total profit of the items that can be placed in a knapsack given that each item has a weight and a profit.
- **Approximation algorithm –** for an *NP*-hard optimization problem is an algorithm that is not guaranteed to give optimal solutions, but rather yields solutions that are reasonably close to optimal.
- **Chromatic number –** the minimum number of colors needed to color a graph so that no two adjacent vertices are colored the same color.
- **Clause –** is a sequence of literals separated by the logical or operator.
- **Clique –** a clique in an undirected graph $G = (V, E)$ is a subset W of V such that each vertex in W is adjacent to all the other vertices in *W.*
- **Clique Decision problem –** determine, for a positive integer *k*, whether there is a clique containing at least *k* vertices.
- **Clique Optimization problem –** is to determine the size of a maximal clique for a given graph.
- **CNF-Satisfiability Decision problem –** determine, for a given logical expression in CNF, whether there is some truth assignment that makes the expression true.
- **Complementary problem –** to a decision problem it is the problem that answers *yes* whenever the original problem answers *no* and answers *no* whenever the original problem answers *yes.*
- **Conjunctive normal form (CNF) –** a sequence of clauses separated by the logical *and* operator
- **Extendible –** to cause to be of greater area or volume.
- **First-fit –** strategy places an item in the first bin in which it fits. If it does not fit in a bin, a new bin is started.
- **Graph-Coloring Decision problem –** is to determine, for an integer *m*, whether there is a coloring that uses at most *m* colors and that colors no two adjacent vertices the same color.
- **Graph-Coloring Optimization problem –** determine the minimum number of colors needed to color a graph so that no two adjacent vertices are colored the same color.
- **Graph Isomorphism problem –** a problem to determine whether two graphs are identical or similar.
- **Hamiltonian Circuits Decision problem –** determine whether a connected, undirected graph has at least one tour.
- **Input size –** the number of characters it takes to write the input.
- **Intractable –** a problem for which no algorithm exists that computes all instances in polynomial time.
- **Literal –** a logical variable or the negation of a logical variable.
- **Logical (Boolean) variable –** is a variable that can have one of two values: true or false.
- **Magnitude –** a numerical quantitative measure, expressed as a multiple of a standard unit.
- **Matching –** a creation of vertex pairs.
- **Maximal clique –** is a clique of maximal size.
- **Minimal weight matching –** the total weight of the edges obtained from the matching is minimal.
- **Nondeterministic algorithm –** a conceptual algorithm with more than one allowed step at certain times and which always takes the right or best step.
- **Nonincreasing first fit** – items packed in non-increasing order
- ***NP*-complete –** a problem in which answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly.
- ***NP*-easy –** A problem A is called NP-easy if, for some problem B in NP, A $\propto_T$ B.
- ***NP*-equivalent –** A problem is called NP-equivalent if it is both *NP*-hard and *NP*-easy.
- ***NP*-hard –** A problem B is called NP-hard if, for some NP-complete problem A, A $\propto_T$ B

- **Polynomial time algorithm –** one whose worst-case time complexity is bounded above by a polynomial function of its input size.
- **Polynomial-time many-one reducible –** if there exists a polynomial-time transformation algorithm from decision problem *A* to decision problem *B*, problem *A* is polynomial-time many one reducible to problem *B*.
- **Polynomial-time nondeterministic algorithm –** a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.
- **Polynomial-time Turing reducible –** if problem *A* can be solved in polynomial time using a hypothetical polynomial time algorithm for problem *B*, then problem *A* is polynomial-time Turing reducible to problem *B*.
- **Pseudopolynomial-time** – an algorithm whose worst-case time complexity is bounded above by a polynomial function of its size and magnitudes.
- **Singly connected** A directed, acyclic graph is singly connected if there is no more than one path from any vertex to any other vertex.
- **Step –** the equivalent of one machine comparison or assignment.
- **Tour –** in a directed graph is a path from a vertex to itself that passes through each of the other vertices exactly once.
- **Traveling Salesperson Optimization problem –** determine a tour with minimal total weight on its edges.
- **Traveling Salesperson Decision problem –** determine for a given positive number d whether there is a tour having total weight no greater than *d*.
- **Triangular inequality –** a complete weighted graph that satisfies weight for all vertices (i.e. there are no short cuts).
- **Undecidable –** problems for which algorithms that can solve them are proven not to exist.
- **Worst-case time complexity –** the maximum number of steps done by the algorithm for an input size of *s*.

Important Links

- www.sciencedirect.com/science/journal/01966774 (Journal of Algorithms)
- http://www.informatik.uni-trier.de/~ley/db/journals/jal/jal5.html
- http://www.seas.gwu.edu/~ayoussef/cs212/npcomplete.html (*NP*-Complete Theory)