بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر

(نیم‌سال تحصیلی ۴۰۰۱)

نظریهٔ زبان‌ها و ماشین‌ها

حسین فلسفین

In Part 2 (Computability Theory) we described the distinction between problems that are theoretically solvable and ones that are not.

In this session, we will take another look at the class of solvable problems and further distinguish among them. In particular, we will contrast problems that are "practically solvable", in the sense that programs that solve them have resource requirements (in terms of time and or space) that can generally be met, and problems that are "practically unsolvable", at least for large inputs, since their resource requirements grow so quickly that they cannot typically be met.

Throughout our discussion, we will generally assume that if resource requirements grow as some polynomial function of problem size, then the problem is practically solvable. If they grow faster than that, then, for all but very small problem instances, the problem will generally be practically unsolvable.

Some computational problems are impractical because the known algorithms to solve them take too much time or space. We will make the idea of "too much" more precise by introducing some fundamental complexity classes of problems.

Decision problems vs. optimization problems

It is more convenient to develop the theory if we originally restrict ourselves to decision problems.

The main results of the theory are stated in terms of decision problems that ask a question whose answer is either Yes or No. So we'll consider only decision problems.

نسخهٔ تصمیم‌گیری و نسخهٔ بهینه‌سازی یک مسئلهٔ معین (مثلاً، مجموعهٔ مستقل بیشینه) با یکدیگر تمایز عمده‌ای ندارند:

In fact, from the point of view of polynomial-time solvability, there is not a significant difference between the optimization version of the problem (find the maximum size of an independent set) and the decision version (decide, yes or no, whether G has an independent set of size at least a given k). Given a method to solve the optimization version, we automatically solve the decision version (for any k) as well. But there is also a slightly less obvious converse to this: If we can solve the decision version of Independent Set for every k, then we can also find a maximum independent set. For given a graph G on n nodes, we simply solve the decision version of Independent Set for each k; the largest k for which the answer is "yes" is the size of the largest independent set in G. (And using binary search, we need only solve the decision version for $O(\log n)$ different values of k.) This simple equivalence between decision and optimization will also hold in the problems we discuss below.

Traveling Salesperson Problem

Let a weighted, directed graph be given. A tour in such a graph is a path that starts at one vertex, ends at that vertex, and visits all the other vertices in the graph exactly once.

☞ *The Traveling Salesperson Optimization problem is to determine a tour with minimal total weight on its edges.*

☞ *The Traveling Salesperson Decision problem is to determine for a given positive number $cost$ whether there is a tour having total weight no greater than $cost$. This problem has the same parameters as the Traveling Salesperson Optimization problem plus the additional parameter $cost$.*

TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than *cost*}.

0-1 Knapsack Problem

☞ *The 0-1 Knapsack Optimization problem is to 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, and that there is a maximum total weight W that can be carried in the sack.*

☞ *The 0-1 Knapsack Decision problem is to 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, while making the total profit at least equal to P. This problem has the same parameters as the 0-1 Knapsack Optimization problem plus the additional parameter P.*

KNAPSACK = $\{<S, v, c>$: S is a set of objects each of which has an associated cost and an associated value, v and c are integers, and there exists some way of choosing elements of S (duplicates allowed) such that the total cost of the chosen objects is at most c and their total value is at least $v\}$.

Graph-Coloring Problem

☞ *The Graph-Coloring Optimization problem is to determine the minimum number of colors needed to color a graph so that no two adjacent vertices are colored the same color. That number is called the chromatic number of the graph.*

☞ *The 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. This problem has the same parameters as the Graph-Coloring Optimization problem plus the additional parameter m.*

$$3COLOR = \{\langle G\rangle \mid G \text{ is colorable with 3 colors}\}.$$

Clique Problem

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.

☞ *The Clique Optimization problem is to determine the size of a maximal clique for a given graph.*

☞ *The Clique Decision problem is to determine, for a positive integer k, whether there is a clique containing at least k vertices. This problem has the same parameters as the Clique Optimization problem plus the additional parameter k.*

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}.$$

CLIQUE = {<G, k> : G is an undirected graph with vertices V and edges E, k is an integer, $1 \le k \le |V|$, and G contains a k-clique}.

بسیار مهم:

There are three general categories of problems as far as intractability is concerned:

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 (NP-complet Problems)*

It is a surprising phenomenon that most problems in computer science seem to fall into either the first or third category.

Class P

Informally, the class P consists of all problems that can be solved in polynomial time. (The class P is the set of all decision problems that can be solved by polynomial-time algorithms.)

The class P consists of those decision problems that can be solved by deterministic algorithms that have worst-case running times of polynomial order. A decision problem is in the class P if there is a deterministic algorithm A that solves the problem and there is a polynomial p such that for each instance I of the problem we have $W_A(n) \leq p(n)$, where n is the size of I. (In short, the class P consists of those decision problems that can be solved by deterministic algorithms of order $O(p(n))$ for some polynomial p.) In other words, polynomial-time algorithm is one whose worst-case time complexity is bounded above by a polynomial function of its input size. That is, if n is the input size, there exists a polynomial $p(n)$ such that $W(n) \in O(p(n))$.

It is common to think of the class P as containing exactly the tractable problems. In other words, it contains those problems that are not only solvable in principle (i.e., they are decidable) but also solvable in an amount of time that makes it reasonable to depend on solving them in real application contexts.

Of course, suppose that the best algorithm we have for deciding some language L is $O(n^{1000})$ (i.e., its running time grows at the same rate, to within a constant factor, as n^{1000}). It is hard to imagine using that algorithm on anything except a toy problem. But the empirical fact is that we don't tend to find algorithms of this sort.

*Most problems of practical interest that are known to be in P can be solved by programs that are **no worse than $O(n^3)$** if we are analyzing running times on conventional (random access) computers. And so they're no worse than $O(n^{18})$ when run on a one-tape, deterministic Turing machine.*

Furthermore, it often happens that, once some polynomial time algorithm is known, a faster one will be discovered. For example, consider the problem of matrix multiplication. If we count steps on a random access computer, the obvious algorithm far matrix multiplication (based on Gaussian elimination) is $O(n^3)$. Strassen's algorithm is more efficient; it is $O(n^{2.81})$. Other algorithms whose asymptotic complexity is even lower (approaching $O(n^2)$) are now known, although they are substantially more complex.

So, as we consider languages that are in P, we will generally discover algorithms whose time requirement is some low-order polynomial function of the length of the input.

Languages That Are in P

There are many familiar problems in the class P.

CONNECTED = $\{<G> : G$ is an undirected graph and G is connected$\}$.

$RELPRIME = \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$.

MST = $\{<G, cost> : G$ is an undirected graph with a positive cost attached to each of its edges and there exists a minimum spanning tree of G with total cost less than $cost\}$.

(Prim Algorithm & Kruskal Algorithm)

PRIMES = $\{w : w$ is the binary encoding of a prime number$\}$.

The Eulerian Circuit Problem

An Eulerian circuit through a graph G is a path that starts at some vertex s, ends back in s, and traverses each edge in G exactly once. (Note the difference between an Eulerian circuit and a Hamiltonian one: An Eulerian circuit visits each edge exactly once. A Hamiltonian circuit visits each vertex exactly once.)

EULERIAN-CIRCUIT = $\{<G> : G$ is an undirected graph and G contains an Eulerian circuit$\}$.

Intractable Problems

Certain computational problems are solvable in principle, but the solutions require so much time or space that they can't be used in practice. Such problems are called intractable.

A problem is said to be tractable if it is in P and intractable if it is not in P. In other words, a problem is intractable if it has a lower-bound worst-case complexity greater than any polynomial. We give examples of problems that we can prove to be intractable.

Oddly enough, we have found relatively few such problems. The first ones were undecidable problems. Of course, any undecidable problem is intractable because there is no algorithm to solve it. So there is no polynomial time algorithm to solve it. For example, the halting problem is intractable. But this isn't very satisfying. So we'll give some real live examples of problems that are intractable.

In computer science, a problem is called intractable if it is impossible to solve it with a polynomial-time algorithm. 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. For example, the brute-force algorithm for the Chained Matrix Multiplication problem is nonpolynomial-time. However, the problem can be solved in $\Theta(n^3)$ using the dynamic programming approach. The problem is not intractable, because we can solve it in polynomial-time.

The dictionary defines intractable as "difficult to treat or work." This means that a problem in computer science is intractable if a computer has difficulty solving it.

A problem for which an efficient algorithm is not possible is said to be "intractable."

Examples

In 1953, A. Grzegorczyk developed a decidable problem that is intractable. Similar results are discussed in Hartmanis and Stearns (1965). However, these problems were "artificially" constructed to have certain properties.

One of the most well-known of these problems is Presburger Arithmetic, which was proven intractable by Fischer and Rabin in 1974. This problem, along with the proof of intractability, can be found in Hopcroft and Ullman (1979).

Presburger Arithmetic

The first intractable problem involves arithmetic formulas. The problem is to decide the truth of statements in a simple theory about addition of natural numbers. The statements of the theory are expressed as closed wffs of a first-order predicate calculus that uses just $+$ *and* $=$*. For example, the following formulas are wffs of the theory:*

$$\forall x\ \forall y(x + y = y + x),$$
$$\exists y\ \forall x(x + y = x),$$
$$\forall x\ \forall y\ \exists z(\neg(x = y) \rightarrow x + z = y),$$
$$\forall x\ \forall y\ \forall z(x + (y + z) = (x + y) + z),$$
$$\forall x\ \forall y(x + x = x \rightarrow x + y = y).$$

Each wff of the theory is either true or false when interpreted over the natural numbers. You might notice that one of the preceding example wffs is false and the other four are true. In 1930, Presburger showed that this theory is decidable. In other words, there is an algorithm that can decide whether any wff of the theory is true. The theory is called Presburger arithmetic. Fischer and Rabin [1974] proved that any algorithm to solve the decision problem for Presburger arithmetic must have an exponential lower bound for the number of computational steps.

یادآوری

Recall that the set of regular expressions over an alphabet A is defined inductively as follows, where $\cup$ and $\circ$ are binary operations and $*$ is a unary operation:

☞ ***Basis:*** *ε, $\varnothing$, and a are regular expressions for all $a \in A$.*

☞ ***Induction:*** *If R and S are regular expressions, then the following expressions are also regular: (R), $R \cup S$, $R \circ S$, and R^*. For example, here are a few of the infinitely many regular expressions over the alphabet $A = \{a, b\}$: ε, $\varnothing$, a, b, $\varepsilon \cup b$, b^*, $a \cup (b \circ a)$, $(a \cup b) \circ a$, $a \circ b^*$, $a^* \cup b^*$.*

Each regular expression represents a regular language. For example, ε represents the language $\{\varepsilon\}$; $\varnothing$ represents the empty language $\varnothing$; $a \circ b^*$ represents the language of all strings that begin with a and are followed by zero or more occurrences of b; and $(a \cup b)^*$ represents the language $\{a, b\}^*$.

Another Example

We show that by allowing regular expressions with more operations than the usual regular operations, the complexity of analyzing the expressions may grow dramatically. Let $\uparrow$ be the exponentiation operation. If R is a regular expression and k is a nonnegative integer, writing $R \uparrow k$ is equivalent to the concatenation of R with itself k times. We also write R^k as shorthand for $R \uparrow k$. In other words,

$$R^k = R \uparrow k = \overbrace{R \circ R \circ \cdots \circ R}^{k}.$$

Generalized regular expressions allow the exponentiation operation in addition to the usual regular operations.

Obviously, these generalized regular expressions still generate the same class of regular languages as do the standard regular expressions because we can eliminate the exponentiation operation by repeating the base expression. Let

$$EQ_{\mathsf{REX}\uparrow} = \{\langle Q, R\rangle \mid Q \text{ and } R \text{ are equivalent regular expressions with exponentiation}\}.$$

$EQ_{\mathsf{REX}\uparrow}$ ***is intractable.***

یک مثال دیگر از کتاب *Hein*:

Suppose we extend the definition of regular expressions to include the additional notation $(R) \uparrow 2$ as an abbreviation for $R \circ R$. For example, we have

$$a \circ a \circ a \circ a \circ a = a \circ ((a) \uparrow 2) \uparrow 2 = a \circ (a \uparrow 2) \circ (a \uparrow 2).$$

A generalized regular expression is a regular expression that may use this additional notation. Now we're in position to state an intractable problem.

☞ *Inequivalence of Generalized Regular Expressions: Given a generalized regular expression R over a finite alphabet A, does the language of R differ from A^*?*

Here are some examples to help us get the idea:

☞ *The language of* $(a \cup b)^*$ *is the same as* $\{a, b\}^*$.

☞ *The language of* $\varepsilon \cup (a \circ b) \uparrow 2 \cup a^* \cup b^*$ *differs from* $\{a, b\}^*$.

☞ *The language of* $\varepsilon \cup (a \circ b) \uparrow 2 \cup (a \cup b)^*$ *is the same as* $\{a, b\}^*$.

Meyer and Stockmeyer [1972] showed that the problem of inequivalence of generalized regular expressions is intractable. They showed it by proving that any algorithm to solve the problem requires exponential space. So it is intractable. We should note that the intractability comes about because we allow abbreviations of the form $(R) \uparrow 2$.

NP-Complete Problems

NP-Complete Problems: thought to be intractable but none that have been proven to be intractable. For example, most people believe the SAT problem and all other NP-complete problems are intractable, although we don't know how to prove that they are.

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. There are many such problems.

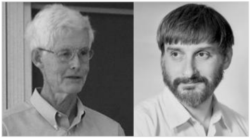
Demonstrating that a language is NP-complete provides strong evidence that the language is not in P.

https://www.claymath.org/millennium-problems/p-vs-np-problem

P vs NP Problem | Clay Mathem

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EVENTS EUCLID

P vs NP Problem



Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Rules:

Rules for the Millennium Prizes

Related Documents:

Official Problem Description

Minesweeper

Related Links:

Lecture by Vijaya Ramachandran

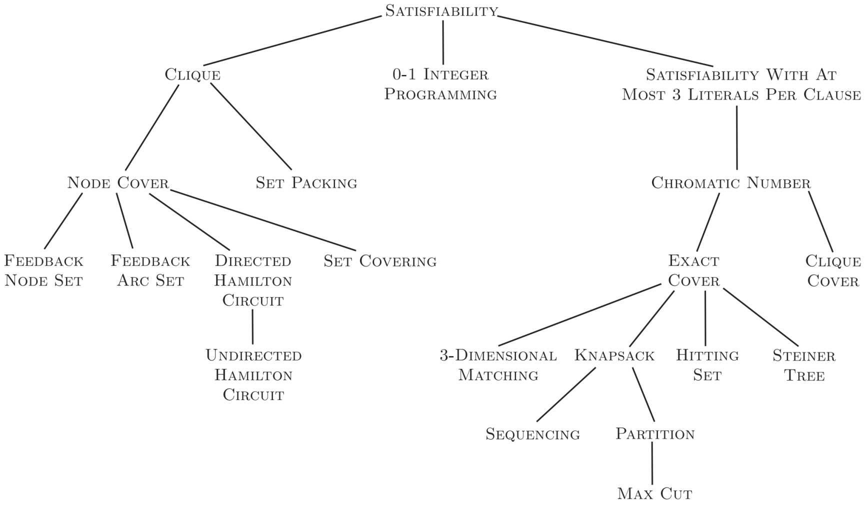
The Cook-Levin Theorem: SAT is NP-complete.

SAT $= \{<w> : w$ is a wff in Boolean logic and w is satisfiable$\}$

Karp's 21 Problems

Karp's paper, entitled "Reducibility among combinatorial problems," showed that in fact NP-completeness isn't rare. Karp explicitly proved that 21 different problems, drawn from a surprising variety of applications in math and computer science, are NP-complete. Thus, he demonstrated that many of the hard problems studied by computer scientists are in some sense equally hard—and that none of them has an efficient method of solution unless $P = NP$.

Karp's 21 Problems



Karp's figure 1, captioned "Complete Problems."

SAT = {<w> : w is a wff in Boolean logic and w is satisfiable}.

3-SAT = {<w> : w is a wff in Boolean logic, w is in 3-conjunctive normal form and w is satisfiable}.

TSP-DECIDE = {<G, *cost*>, where <G> encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than *cost*}.

HAMILTONIAN-PATH = {<G> : G is an undirected graph and G contains a Hamiltonian path}.

HAMILTONIAN-CIRCUIT = {<G> : G is an undirected graph and G contains a Hamiltonian circuit}.

CLIQUE = {<G, k> : G is an undirected graph with vertices V and edges E, k is an integer, $1 \leq k \leq |V|$, and G contains a k-clique}.

INDEPENDENT-SET = {<G, k> : G is an undirected graph and G contains an independent set of at least k vertices}.

SUBSET-SUM = $\{<S, k> : S$ is a multiset (i.e., duplicates are allowed) of integers, k is an integer, and there exists some subset of S whose elements sum to $k\}$.

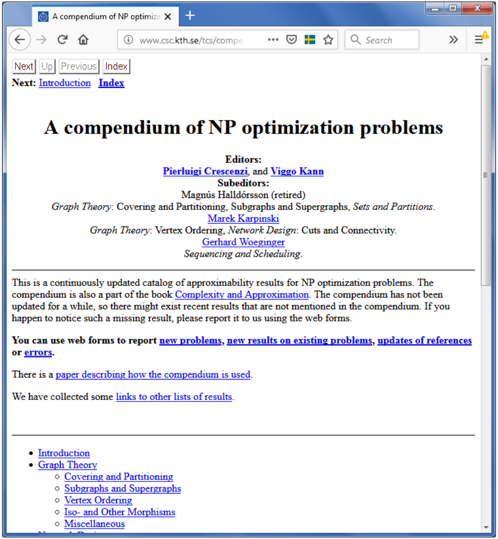
SET-PARTITION = $\{<S> : S$ is a multiset (i.e., duplicates are allowed) of objects each of which has an associated cost and there exists a way to divide S into two subsets, A and $S - A$, such that the sum of the costs of the elements in A equals the sum of the costs of the elements in $S - A\}$.

KNAPSACK = $\{<S, v, c> : S$ is a set of objects each of which has an associated cost and an associated value, v and c are integers, and there exists some way of choosing elements of S (duplicates allowed) such that the total cost of the chosen objects is at most c and their total value is at least $v\}$.

SUDOKU = $\{<b> : b$ is a configuration of an $n \times n$ Sudoku grid and b has a solution$\}$.

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

http://www.csc.kth.se/tcs/compendium/



A compendium of NP optimiza ×

www.csc.kth.se/tcs/compe

Search

Next | Up | Previous | Index

Next: Introduction **Index**

A compendium of NP optimization problems

Editors:
Pierluigi Crescenzi, and **Viggo Kann**
Subeditors:
Magnús Halldórsson (retired)
Graph Theory: Covering and Partitioning, Subgraphs and Supergraphs, *Sets and Partitions*.
Marek Karpinski
Graph Theory: Vertex Ordering, *Network Design*: Cuts and Connectivity.
Gerhard Woeginger
Sequencing and Scheduling.

This is a continuously updated catalog of approximability results for NP optimization problems. The compendium is also a part of the book Complexity and Approximation. The compendium has not been updated for a while, so there might exist recent results that are not mentioned in the compendium. If you happen to notice such a missing result, please report it to us using the web forms.

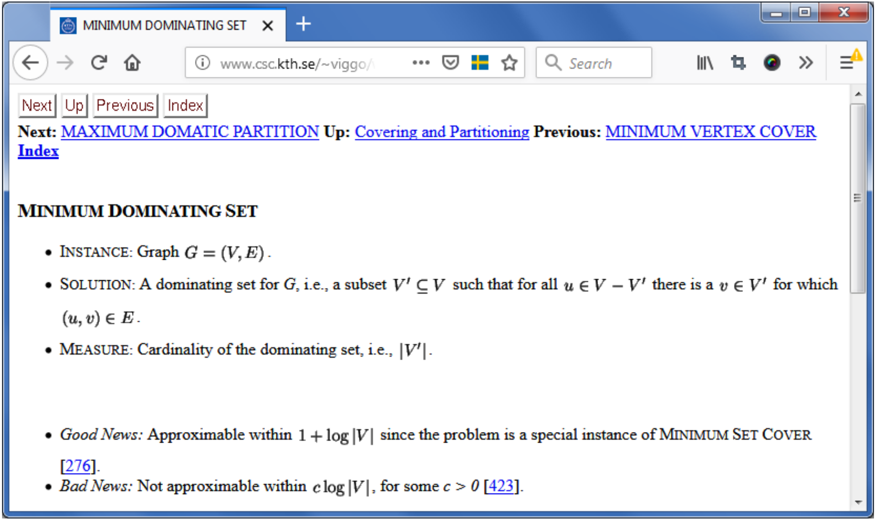
You can use web forms to report new problems, new results on existing problems, updates of references or errors.

There is a paper describing how the compendium is used.

We have collected some links to other lists of results.

- Introduction
- Graph Theory
  - Covering and Partitioning
  - Subgraphs and Supergraphs
  - Vertex Ordering
  - Iso- and Other Morphisms
  - Miscellaneous

http://www.csc.kth.se/tcs/compendium/



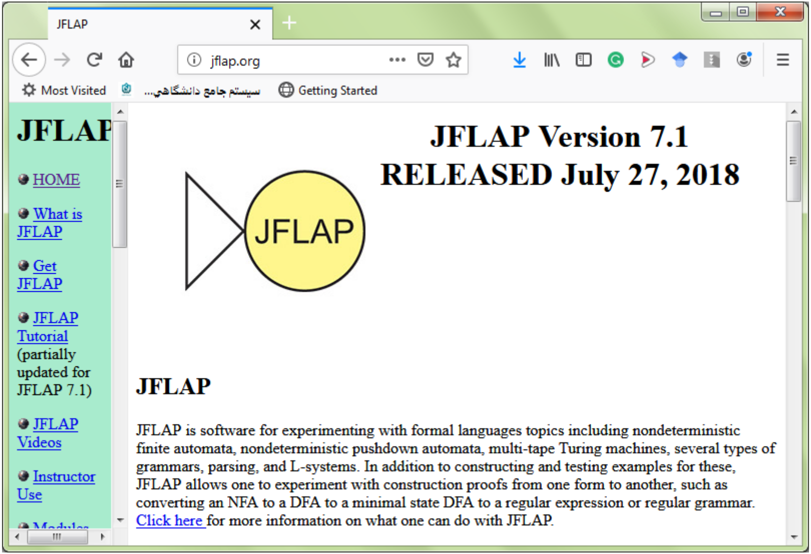
Next Up Previous Index

Next: MAXIMUM DOMATIC PARTITION **Up:** Covering and Partitioning **Previous:** MINIMUM VERTEX COVER
Index

MINIMUM DOMINATING SET

- INSTANCE: Graph $G = (V, E)$.
- SOLUTION: A dominating set for G, i.e., a subset $V' \subseteq V$ such that for all $u \in V - V'$ there is a $v \in V'$ for which $(u, v) \in E$.
- MEASURE: Cardinality of the dominating set, i.e., $|V'|$.

- *Good News:* Approximable within $1 + \log |V|$ since the problem is a special instance of MINIMUM SET COVER [276].
- *Bad News:* Not approximable within $c \log |V|$, for some $c > 0$ [423].


JFLAP
jflap.org
Most Visited
سیستم جامع دانشگاهي...
Getting Started
JFLAP
HOME
What is JFLAP
Get JFLAP
JFLAP Tutorial (partially updated for JFLAP 7.1)
JFLAP Videos
Instructor Use
JFLAP Version 7.1
RELEASED July 27, 2018
JFLAP
JFLAP
JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar.
Click here for more information on what one can do with JFLAP.