DECIDABILITY AND UNDECIDABILITY

Decidable problems from language theory

For simple machine models, such as finite automata or pushdown automata, many decision problems are solvable. In the case of deterministic finite automata, problems like equivalence can be solved even in polynomial time. Also there are efficient parsing algorithms for context-free grammars.

If necessary we may briefly review some material on regular and context-free languages from chapters 1 and 2 in the textbook. Recall in particular the following important characterization:

Regular languages = languages denoted by regular expressions
= languages accepted by DFAs (deterministic finite automata)
= languages accepted by NFAs (nondeterministic finite automata).

The class of regular languages is strictly contained in the deterministic context-free languages (DCFL) which in turn are strictly contained in the (general) context-free languages. The class DCFL consists of languages recognized by deterministic pushdown automata.

We recall the following basic notions. A decision problem is a restricted type of an algorithmic problem where for each input there are only two possible outputs.

- A *decision problem* is a function that associates with each input instance of the problem a truth value *true* or *false*.
- A *decision algorithm* is an algorithm that computes the correct truth value for each input instance of a decision problem. The algorithm has to terminate on all inputs!
- A decision problem is *decidable* if there exists a decision algorithm for it. Otherwise it is *undecidable*.

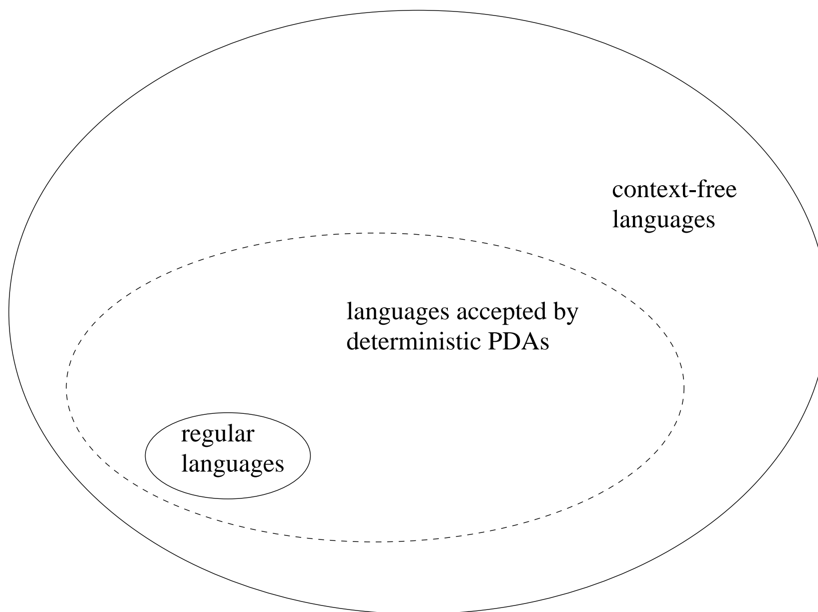


Figure 1: Regular, context-free and deterministic context-free languages

To show that a decision problem is decidable it is sufficient to give an algorithm for it. On the other hand, how could we possibly establish (= prove) that some decision problem is undecidable? This is one of the questions we will address in the course.

Decidability properties of regular languages

Important decision problems for finite automata include the following:

1. DFA membership

   INSTANCE: A DFA $M = (Q, \Sigma, \delta, q_0, F)$ and a string $w \in \Sigma^*$

   QUESTION: Is $w \in L(M)$?

   Proposition. DFA membership is decidable.

   Proof. To be explained in class: the algorithm simulates the given DFA on the given input.

2. DFA emptiness.

INSTANCE: A DFA $M = (Q, \Sigma, \delta, q_0, F)$

QUESTION: Is $L(M) = \emptyset$?

Theorem. DFA emptiness is decidable.

Proof. We note that $L(M) = \emptyset$ iff there is no path in the state diagram of M from q_0 to a final state. If $F = \emptyset$, then clearly $L(M) = \emptyset$. Otherwise, we use a graph reachability algorithm to enumerate all states that can be reached from q_0 and check whether this set contains some state of F. The algorithm terminates because the state diagram is finite.

(This result is explained in Ch. 4 of the textbook.)

3. DFA universality

INSTANCE: A DFA $M = (Q, \Sigma, \delta, q_0, F)$

QUESTION: Is $L(M) = \Sigma^*$?

Theorem. DFA universality is decidable.

Proof: in class

4. DFA containment

INSTANCE: Two DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M = (Q_2, \Sigma, \delta_2, q_2, F_2)$

QUESTION: Is $L(M_1) \subseteq L(M_2)$?

Theorem. DFA containment is decidable.

Proof hint: using closure properties of regular languages reduce the question to checking emptiness.

5. DFA equivalence

INSTANCE: Two DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M = (Q_2, \Sigma, \delta_2, q_2, F_2)$

QUESTION: Is $L(M_1) = L(M_2)$?

Theorem. DFA equivalence is decidable.

Proof hint: reduce the question to checking containment (see section 4.1).

Regular languages are useful for many practical applications due to the fact that "all natural" questions concerning regular languages are decidable.[1] The downside is that the family of regular languages is quite small.

As we will see, already for context-free languages some of the above questions are *undecidable* (universality, containment, equivalence).

For languages accepted by general Turing machines, as we will shortly find out, *all non-trivial questions are undecidable!*

Also the corresponding decision problems for context-free grammars are discussed in section 4.1.

Context-free membership A_{CFG}

INSTANCE: A CF grammar $G = (V, \Sigma, R, S)$ and a string $w \in \Sigma^*$.

QUESTION: Is $w \in L(G)$?

Formally, A_{CFG} is defined to be the language consisting of all encodings[2] of a pair consisting of a CFG and some string generated by the grammar:

$$A_{CFG} = \{< G, w > \mid G \text{ is a CFG and } w \in L(G)\}$$

Theorem. A_{CFG} is decidable.

[1] There are known exceptions, but these are somewhat "artificial" problems.

[2] Inputs to TMs must be encoded as strings. The notation for the encoding is explained at the end of chapter 3 in the textbook.

Note: In the context of computability theory, to show that A_{CFG} is decidable it is sufficient to use a simple brute-force parsing algorithm. Context-free grammars can be parsed efficiently and the best known parsing algorithms for general context-free grammars have time complexity (slightly less than) $O(n^3)$.

Similarly, the context-free emptiness problem is encoded as a language:

$$E_{CFG} = \{< G >| \ G \text{ is a CFG and } L(G) = \emptyset\}$$

Theorem. E_{CFG} is decidable.

Proof. In class.

Also, we can consider the equivalence problem for context-free languages. Formally this can be encoded as the language

$$EQ_{CFG} = \{< G, H >| \ G, H \text{ are CFGs and } L(G) = L(H)\}$$

Recall that we have an algorithm that decides equivalence of DFAs or NFAs. However, the same approach that was used to establish the decidability result for DFAs does not work if we try to show that EQ_{CFG} is decidable. Why not?

In fact, it turns out that EQ_{CFG} is not decidable, that is, the equivalence problem for context-free grammars is undecidable! We will come back to this later[3].

Undecidable problems

We will now discuss the notion of undecidability. This is section 4.2 in the textbook. First let us review some terminology.

[3] The above observations that our previous method cannot be used to establish decidability still in no way guarantee that the question is undecidable.

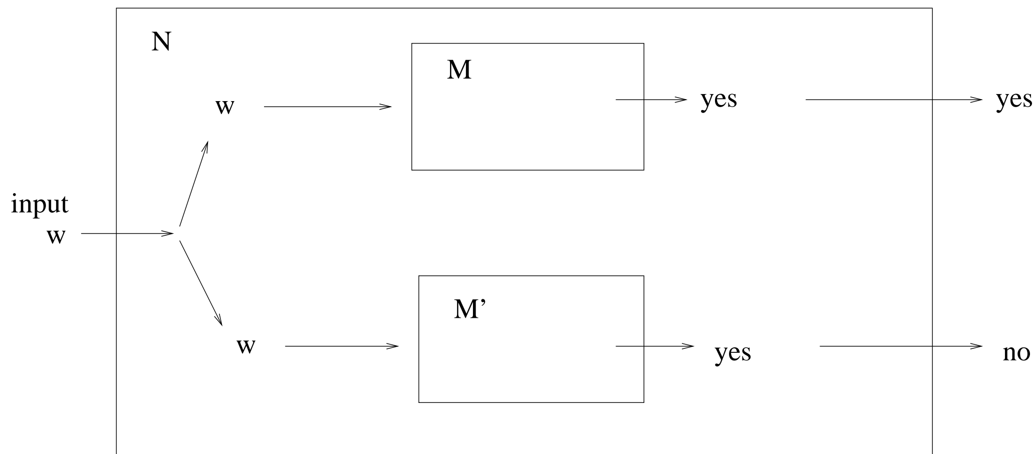


Figure 2: A decider for the language L.

- A language is *decidable* if some TM decides it (chapter 3). All computations of a decider TM must halt. Decidable languages are often called also *recursive languages.*

- A language is *Turing-recognizable* (or *recursively enumerable*) if it is recognized by a TM. That is, all words in the language are accepted by the TM. On words not belonging to the language, the computation of the TM either rejects or goes on forever.

Lemma. A language L is decidable if and only if $\bar{L}$ is decidable.

Proof: in class

Theorem. L is decidable if and only if both L and $\bar{L}$ are Turing-recognizable.

Proof. *(only if):* Follows from the previous lemma and the fact that every decidable language is Turing-recognizable.

(if): Let M be a TM recognizing L and M' a TM recognizing $\bar{L}$. We construct a decider N for the language L, see Figure 2.

The decider N can be implemented as a 2-tape TM that on its first tape simulates M and "in parallel" on the second tape simulates M'. If the simulation on tape one ac-

cepts, N accepts. If the simulation on tape two accepts, N rejects. One of the simulations necessarily halts in a finite number of steps. (Why?)

The standard example of an undecidable language is:

$$L_{TMaccept} = \{< M, w >| \ M \text{ is a TM and } M \text{ accepts } w\}$$

Theorem. $L_{TMaccept}$ is undecidable.

The proof (to be gone through in class) shows that, in fact, the more restricted language

$$L_{selfaccept} = \{< M, < M >>| \ M \text{ is a TM } \}$$

is undecidable. The crucial idea is diagonalization.

Universal Turing machines

General purpose computers operate as follows:

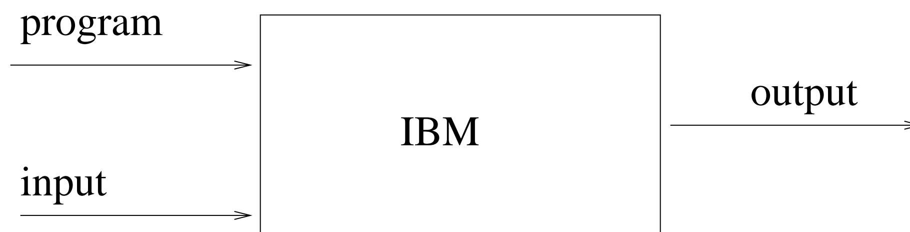


Figure 3: Programmable computer

Similarly, can view a universal Turing-machine to be "programmable":

$< M >$: encoding of TM M;

$< x >$: encoding of input x to M;

$< y >$: encoding of output produced by M.

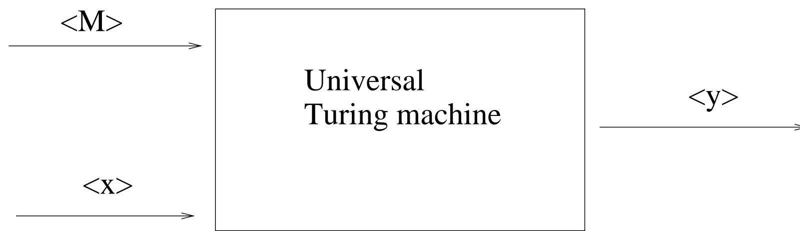


Figure 4: "Programmable" (universal) Turing-machine

The proof of the existence of universal TMs is *constructive.*

Outline of the proof:

- We use three tapes:

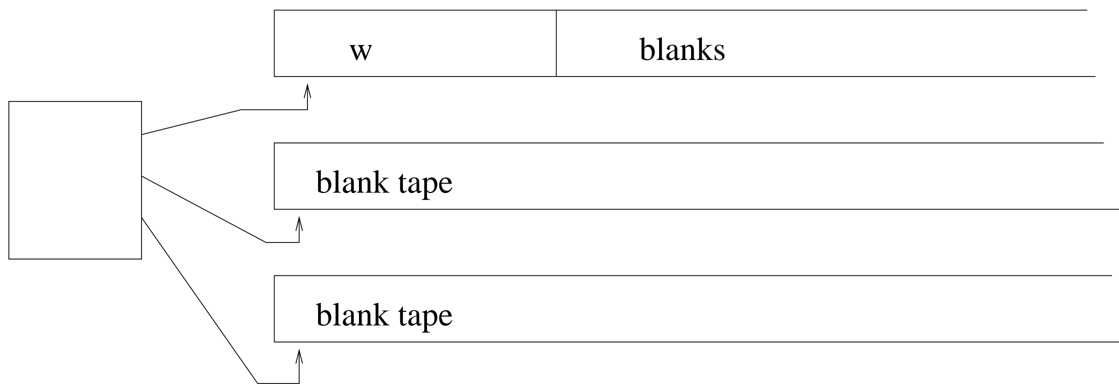


Figure 5: The configuration at the beginning, here $w =< M, x >$.

- Steps:

1. Check the validity of the input (correct encoding of a TM M and an input x for M).
2. Copy from the input tape the string x to the second tape.
3. Write the start state of M onto third tape.
4. Start the simulation, see Figure 6.

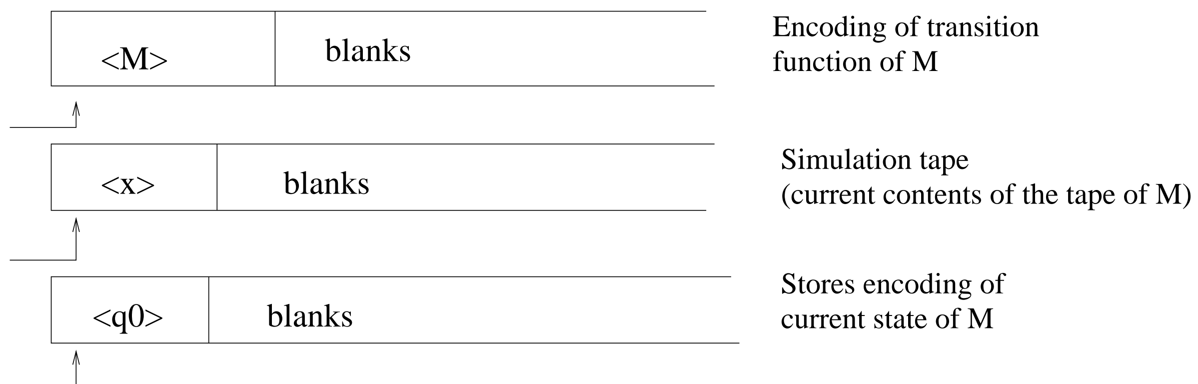


Figure 6: The contents of the tapes after steps 1., 2., 3. After this the universal machine is ready to begin the simulation of M.

Note:

- A universal TM has a fixed tape alphabet.
- Different TMs have different state sets and tape alphabets, and these may be arbitrarily large finite sets.

Consequently the TMs given as input for a universal TM must be encoded using a fixed alphabet. The encoding must include the state set, tape alphabet and transition function. This is illustrated in the below example.

Example.

The TM of Figure 7 could be first encoded as a string:

$[\delta(0, a) = (1, b, R);\ \delta(1, b) = (0, b, R);\ \delta(1, \sqcup) = (2, \sqcup, L);$ 1s; 2acc$]$

Above "1s" would denote that "1" is the start state and "2acc" denotes that "2" is the accept state.

The above string could then straightforwardly be encoded using a binary alphabet. In this way, any Turing machine can be encoded as a string over the binary alphabet.

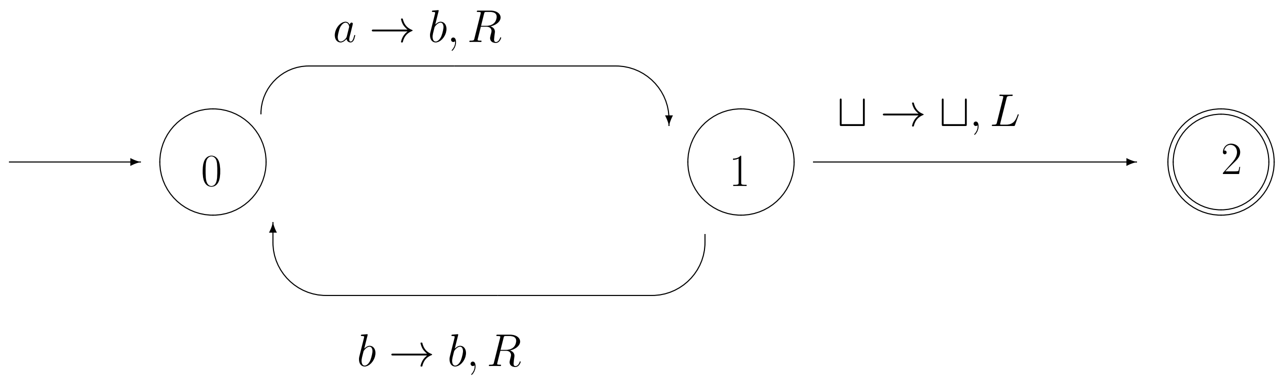


Figure 7: An example of a Turing machine.

- There exist fairly small universal TMs. For example, we can construct a universal TM that has 7 states and the tape alphabet has 4 symbols.

- There do not exist "universal DFAs", that is, DFAs that could simulate any other DFA. Why not?

The language

$$L_{TMaccept} = \{< M, w > \mid M \text{ is a TM and } M \text{ accepts } w\}$$

is recognized by a universal TM!

- This shows that there exist Turing-recognizable languages that are not decidable.

- On the other hand, the complement of $L_{TMaccept}$ is not Turing-recognizable. Why not?

Post Correspondence Problem (PCP)

INPUT: $\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\}$

PROBLEM: Is there $i_1, i_2, \ldots, i_m$ such that $t_{i_1} t_{i_2} \ldots t_{i_m} = b_{i_1} b_{i_2} \ldots b_{i_m}$?

$t_1, t_2, \ldots, t_k, b_1, b_2, \ldots, b_k$ are strings over some alphabet Σ.

A solution, if it exists, is called a *match*.

PCP - example

For the collection of dominos below:

$$\left[\frac{ab}{aba}\right], \left[\frac{ba}{abb}\right], \left[\frac{b}{ab}\right], \left[\frac{abb}{b}\right], \left[\frac{a}{bab}\right]$$

here is a match:

$$\left[\frac{ab}{aba}\right] \left[\frac{a}{bab}\right] \left[\frac{ba}{abb}\right] \left[\frac{b}{ab}\right] \left[\frac{abb}{b}\right] \left[\frac{abb}{b}\right] \left[\frac{b}{ab}\right] \left[\frac{abb}{b}\right]$$

For the collection of dominos below:

$$\left[\frac{ab}{aba}\right], \left[\frac{ba}{abb}\right], \left[\frac{b}{ab}\right]$$

there is no match.

Modified Post Correspondence Problem (MPCP)

Require that the match start with the first domino.

INPUT: $\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\}$

PROBLEM: Is there $i_2, \ldots, i_m$ such that $t_1 t_{i_2} \ldots t_{i_m} \; = \; b_1 b_{i_2} \ldots b_{i_m}$?

$t_1, t_2, \ldots, t_k, b_1, b_2, \ldots, b_k$ are strings over some alphabet Σ.

Reduction from MPCP to PCP

Let $u = u_1 u_2 \ldots u_n$ be a string. Define

$$*u = *u_1 * u_2 * \ldots * u_n.$$

$$u* = u_1 * u_2 * \ldots * u_n *.$$

$$*u* = *u_1 * u_2 * \ldots * u_n *.$$

Given the collection of dominos:

$$\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\}$$

output the collection of dominos:

$$\{\left[\frac{*t_1}{*b_1*}\right], \left[\frac{*t_1}{b_1*}\right], \left[\frac{*t_2}{b_2*}\right], \ldots, \left[\frac{*t_k}{b_k*}\right], \left[\frac{*\diamond}{\diamond}\right]\}.$$

Reduction from A_{TM} to MPCP

Given $M = (Q, \Sigma, \Gamma, \delta, q_s, q_{acc}, q_{rej})$ and $w = w_1 w_2 \ldots w_n$ a reduction machine constructs dominos as described below:

1. $\left[\frac{\#}{\# q_s w_1 w_2 \ldots w_n \#}\right]$.

2. For all $a, b \in \Gamma$, for all $q, r \in Q$ so that $q \neq q_{rej}$:

   If $\delta(q, a) = (r, b, R)$ add dominos $\left[\frac{qa}{br}\right]$.

3. For all $a, b, c \in \Gamma$, for all $q, r \in Q$ so that $q \neq q_{rej}$:

   If $\delta(q, a) = (r, b, L)$ add dominos $\left[\frac{cqa}{rcb}\right]$.

4. For all $a \in \Gamma$, add dominos $\left[\frac{a}{a}\right]$.

5. Add dominos $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup \#}\right]$.

6. For all $a \in \Gamma$, add dominos $\left[\frac{a q_{acc}}{q_{acc}}\right]$ and $\left[\frac{q_{acc} a}{q_{acc}}\right]$.

7. Add domino $\left[\frac{q_{acc} \# \#}{\#}\right]$

Correctness

- Any solution must begin with $\left[\frac{\#}{\# q_s w_1 w_2 \dots w_n \#}\right]$.
- q_{acc} is not in the dominos: bottom string longer than top string.
- Growing the top part makes the bottom part represent the next configuration:

$$\frac{\alpha\#}{\alpha\#x\#} \longrightarrow \frac{\alpha\#x\#}{\alpha\#x\#y\#}$$

  (y is the configuration next to x.)

- If M does not accept w, q_{acc} never appears in the bottom.

  The lengths are always different and hence no match.

Tractable & Intractable Problems

- We will be looking at :
  - What is a P and NP problem
  - NP-Completeness
  - The question of whether P=NP
  - The Traveling Salesman problem again

Polynomial Time (P)

- Most of the algorithms we have looked at so far have been **polynomial-time algorithms**
- On inputs of size n, their worst-case running time is $O(n^k)$ for some constant k
- The question is asked can all problems be solved in polynomial time?
- From what we've covered to date the answer is obviously no. There are many examples of problems that cannot be solved by any computer no matter how much time is involved
- There are also problems that can be solved, but not in time $O(n^k)$ for any constant k

Non-Polynomial Time (NP)

- Another class of problems are called NP problems
- These are problems that we have yet to find efficient algorithms in Polynomial Time for, but given a solution we can verify that solution in polynomial time
- Can these problems be solved in polynomial time?
- It has not been proved if these problems can be solved in polynomial time, or if they would require superpolynomial time
- This so-called P != NP question is one which is widely researched and has yet to be settled

Tractable and Intractable

- Generally we think of problems that are solvable by polynomial time algorithms as being tractable, and problems that require superpolynomial time as being intractable.
- Sometimes the line between what is an ‘easy’ problem and what is a ‘hard’ problem is a fine one.
- For example, “Find the shortest path from vertex x to vertex y in a given weighted graph”. This can be solved efficiently without much difficulty.
- However, if we ask for the longest path (without cycles) from x to y, we have a problem for which no one knows a solution better than an exhaustive search

Deterministic v Non-Deterministic

- Let us now define some terms
  - **P:** The set of all problems that can be solved by deterministic algorithms in polynomial time

- By *deterministic* we mean that at any time during the operation of the algorithm, there is only one thing that it can do next

- A nondeterministic algorithm, when faced with a choice of several options, has the power to "guess" the right one.

- Using this idea we can define NP problems as,
  - **NP:**The set of all problems that can be solved by nondeterministic algorithms in polynomial time.

Is P = NP?

- Obviously, any problem in P is also in NP, but not the other way around
- To show that a problem is in NP, we need only find a polynomial-time algorithm to check that a given solution (the guessed solution) is valid.
- But the idea of nondeterminism seems silly. A problem is in NP if we can 'guess' a solution and verify it in polynomial time!!
- No one has yet been able to find a problem that can be proven to be in NP but not in P
- Is the set P = NP? We don't know. If it is, then there are many efficient algorithms out there just waiting to be discovered.
- Most researchers believe that P != NP, but a proof remains to be shown

NP-Completeness

- **NP-complete** problems are set of problems that have <u>been proved</u> to be in NP
- That is, a nondeterministic solution is quite trivial, and yet no polynomial time algorithm has yet been developed.
- This set of problems has an additional property which does seem to indicate that P = NP
- If any of the problems can be solved in polynomial time on a deterministic machine, then all the problems can be solved in NP(Cook's Theorem)
- It turns out that many interesting practical problems have this characteristic

Examples of NP-Complete

- **Travelling Salesman Problem**: Given a set of cities and distances between all pairs, find a tour of all the cities of distance less than M.
- **Hamiltonian Cycle**: Given a graph, find a simple cycle that includes all the vertices.
- **Partition**: Given a set of integers, can they be divided into two sets whose sum is equal?
- **Integer Linear Programming**: Given a linear program is there an integer solution?
- **Vertex Cover**: Given a graph and an integer N, is there a set of fewer than N vertices which touches all the edges?

Solving These Problems

- At present no algorithms exist that are guaranteed to solve any of the NP-complete problems efficiently
- Remember if we could find one then we could solve all the NP-Complete problems
- In the meantime can we find ‘adequate’ solutions?
- One approach is to seek an approximate solution which may not be the optimal but is close to the optimal
- Another approach is to focus on the average case and develop an algorithm that works for most, but not all, cases

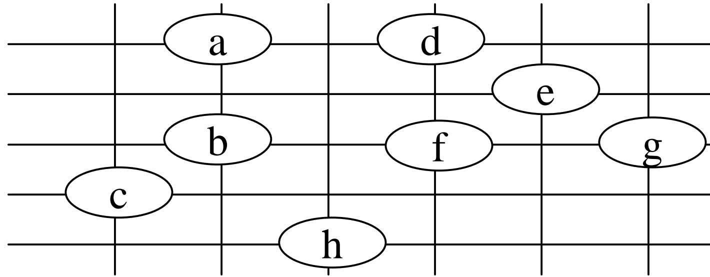
Approximation Algorithms

- Here is an approximation algorithm for the travelling salesman problem,

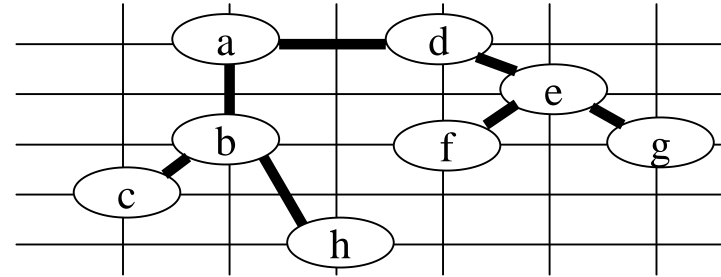
```
Approx-TSP-Tour(G, c)
        - select a vertex r ∈ V[G] to be a "root" vertex
        - grow a minimum spanning tree T for G from root r using MST-Prim(G,c,r)
        - Let L be the list of vertices visited in a preorder tree walk of T
        - return the Hamiltonian cycle H that visits the vertices in the order L
endalg
```

- This approximation if implemented correctly returns a tour whose cost is not more than twice the cost of an optimal tour

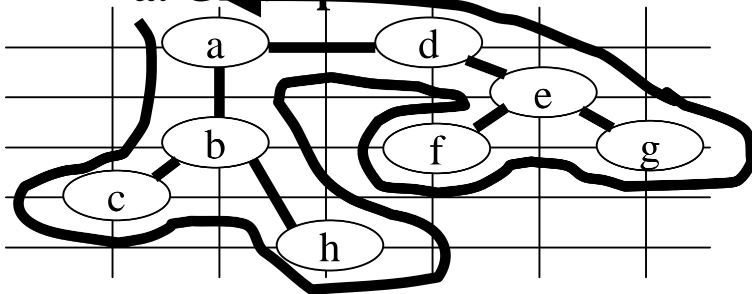
TSP Example



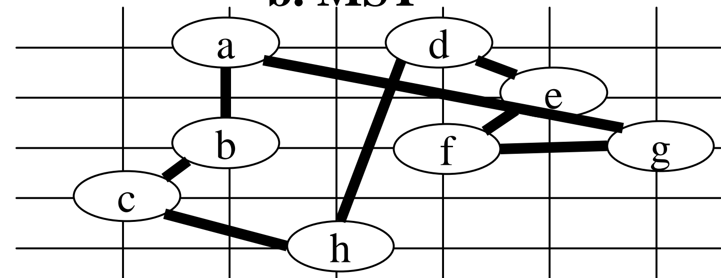
a. Given points



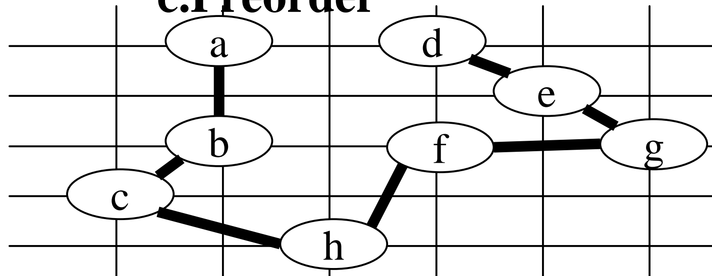
b. MST



c.Preorder



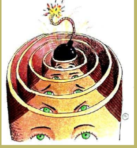
d. Tour by preorder



e. Optimum Tour

Summary

- Polynomial problems are problems for which algorithms can be found that solve the problem in polynomial time
- Non deterministic Polynomial problems are problems for which non-deterministic algorithms can be found
- NP-Complete problems are problems that are in NP, but which have the added property that if one can be solved, they all can be solved
- It is thought that NP-Complete problems may be insoluble using deterministic methods
- One approach is to develop approximation algorithm, as we did with TSP



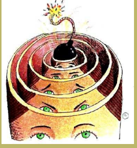
Tractable and . . .

Solving Intractable . . .

Lecture 29:
Tractable and Intractable Problems

Aims:

- To look at the ideas of
  - polynomial and exponential functions and algorithms; and
  - tractable and intractable problems.
- To look at ways of solving intractable problems.



29.1. Tractable and Intractable Problems

- Let's start by reminding ourselves of some common functions, ordered by how fast they grow.

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
n-log-n	$O(n \times \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(k^n)$, e.g. $O(2^n)$
factorial	$O(n!)$
super-exponential	e.g. $O(n^n)$

- Computer Scientists divide these functions into two classes:

Polynomial functions: Any function that is $O(n^k)$, i.e. bounded from above by n^k for some constant k.

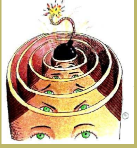
E.g. $O(1)$, $O(\log n)$, $O(n)$, $O(n \times \log n)$, $O(n^2)$, $O(n^3)$

This is really a different definition of the word 'polynomial' from the one we had in a previous lecture. Previously, we defined 'polynomial' to be any function of the form $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$.

But here the word 'polynomial' is used to lump together functions that are bounded from above by polynomials. So, $\log n$ and $n \times \log n$, which are not polynomials in our original sense, are polynomials by our alternative definition, because they are bounded from above by, e.g., n and n^2 respectively.

Exponential functions: The remaining functions.

E.g. $O(2^n)$, $O(n!)$, $O(n^n)$



This is a real abuse of terminology. A function of the form k^n is genuinely exponential. But now some functions which are worse than polynomial but not quite exponential, such as $O(n^{\log n})$, are also (incorrectly) called exponential. And some functions which are worse than exponential, such as the super-exponentials, e.g. $O(n^n)$, will also (incorrectly) be called exponential. A better word than 'exponential' would be 'super-polynomial'. But 'exponential' is what everyone uses, so it's what we'll use.

- Why have we lumped functions together into these two broad classes? The next two tables and the graph attempt to show why.

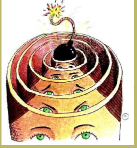
	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n \times \log n$	33	282	665	2469	9966
n^2	100	2500	10000	90000	1 million (7 digits)
n^3	1000	125000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)
2^n	1024	a 16-digit number	a 31-digit number	a 91-digit number	a 302-digit number
$n!$	3.6 million (7 digits)	a 65-digit number	a 161-digit number	a 623-digit number	unimaginably large
n^n	10 billion (11 digits)	an 85-digit number	a 201-digit number	a 744-digit number	unimaginably large

(The number of protons in the known universe has 79 digits.)
(The number of microseconds since the Big Bang has 24 digits.)

(Note that this graph has logarithmic axes.)

- On the basis of this classification of functions into polynomial and exponential, we can classify algorithms:

  Polynomial-Time Algorithm: an algorithm whose order-of-magnitude time performance is bounded from above by a polynomial function of n, where n is the size of its inputs.



Exponential Algorithm: an algorithm whose order-of-magnitude time performance is not bounded from above by a polynomial function of n.

- Why do we divide algorithms into these two broad classes? The next table, which assumes that one instruction can be executed every microsecond, attempt to show you why.

	10	20	50	100	300
n^2	$\frac{1}{10000}$ second	$\frac{1}{2500}$ second	$\frac{1}{400}$ second	$\frac{1}{100}$ second	$\frac{9}{100}$ second
n^5	$\frac{1}{10}$ second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
2^n	$\frac{1}{1000}$ second	1 second	35.7 years	400 trillion centuries	a 75-digit number of centuries
n^n	2.8 hours	3.3 trillion years	a 70-digit number of centuries	a 185-digit number of centuries	a 728-digit number of centuries

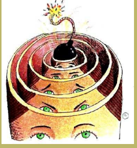
(The Big Bang was approximately 15 billion years ago.)

- And, in a similar way, we can classify problems into two broad classes:

Tractable Problem: a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.

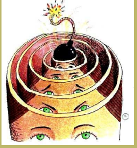
Intractable Problem: a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

- Here are examples of tractable problems (ones with known polynomial-time algorithms):
  - Searching an unordered list



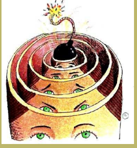
- Searching an ordered list
- Sorting a list
- Multiplication of integers (even though there's a gap)
- Finding a minimum spanning tree in a graph (even though there's a gap)

- Here are examples of intractable problems (ones that have been proven to have no polynomial-time algorithm).
  - Some of them require a non-polynomial amount of output, so they clearly will take a non-polynomial amount of time, e.g.:
    - Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.
    - List all permutations (all possible orderings) of n numbers.
  - Others have polynomial amounts of output, but still cannot be solved in polynomial time:
    - For an $n \times n$ draughts board with an arrangement of pieces, determine whether there is a winning strategy for White (i.e. a sequence of moves so that, no matter what Black does, White is guaranteed to win). We can prove that any algorithm that solves this problem must have a worst-case running time that is at least 2^n.
- So you might think that problems can be neatly divided into these two classes. But this ignores 'gaps' between lower and upper bounds. Incredibly, there are problems for which the state of our knowledge is such that the gap spans this coarse division into tractable and intractable. So, in fact, there are three broad classes of problems:
  - Problems with known polynomial-time algorithms.
  - Problems that are provably intractable (proven to have no polynomial-time algorithm).



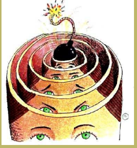
– Problems with no known polynomial-time algorithm but not yet proven to be intractable.

We'll see some examples of the third category (as well as further examples of the first two categories) in the next lecture.



29.2. Solving Intractable Problems

- None of this would matter much if the problems for which we do not have polynomial-time algorithms were theoretical curiosities. Unfortunately, this is not the case. Many real-world problems fall into this category. Unless your inputs are going to be very small, you cannot simply use the known algorithms.
- So what do you do if your problem
  - is provably intractable (proven to have no polynomial-time algorithm), or
  - has no known polynomial-time algorithm even if it is not yet proven intractable?
- Here are the main possibilities:
  - Seek to obtain as much improvement as possible and live hopefully! For example, our backtracking solution to n-Queens was probably better than our first solution. Eliminating symmetry in the problem may help further. Incorporating rules-of-thumb ('heuristics') to dynamically decide what to try next may also help. All of these ideas try to make the algorithm work well in practice, on typical instances, while acknowledging that exponential cases are still possible.
  - Solve simpler/restricted versions of the problem. Maybe a solution to a slight variant of the problem would still be useful to you, while possibly avoiding exponential complexity.
  - Use a polynomial-time probabilistic algorithm: one which gives the right answer only with very high probability. So you are giving up on program correctness, in the interests of speed.
  - For optimisation problems, use a polynomial-time approximation algorithm: one which is not guaranteed to find the best answer.



Tractable and . . .

Solving Intractable . . .

Module Home Page

Title Page

Back

Full Screen

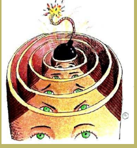
Close

Quit

Acknowledgements:

The tables and graphs come from [Har92].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.



References

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.