Polynomial time algorithms and problems

A key distinction can be made between algorithms that have polynomial time performance and ones that don't.

Intractable algorithms

An algorithm with worst-case time complexity $\Omega(2^N)$ will (for large enough input) take more time that any algorithm with worst case time complexity in $O(N^k)$ for any k, even, e.g. $O(N^{100})$.

We call an algorithm *intractable* if there is no fixed k such that its worst-case time complexity is $O\left(N^k\right)$.

Tractable algorithms

An algorithm with worst case time complexity in $O(N^k)$ for small fixed k will finish in a "reasonable" amount of time on "reasonable" sized inputs.

We call algorithms *tractable* if their worst-case time is in $O(N^k)$ for some fixed k.

In practice k is usually ≤ 6. So "tractable $\simeq$ practical"

A Table

The following table assumes each operation takes $1\,\mathrm{ns}$. Recall that the universe is about 15×10^9 y old.

	N=10	N=50	N=100	N=1000
N	10 ns	50 ns	100 ns	1 μs
$N \log_2 N$	33 ns	282 ns	664 ns	10 μs
N^2	100 ns	2.5 μs	10 μs	1 ms
N^3	1 μs	125 μs	1 ms	1 s
N^{100}	3×10^{83}y	2.5×10^{179}y	3×10^{209}y	3×10^{310}y
1.1^N	2.6 ns	117 ns	13 μs	8×10^{50}y
2^N	1 μs	3.5×10^{24}y	4×10^{39}y	3×10^{310}y
$N!$	3 ms	10×10^{73}y	3×10^{167}y	1.3×10^{2577}y
2^{2^N}	6×10^{317}y	big	Bigger	HUGE

Note that polynomial time algorithms with large exponents are not common.

And exponential time algorithms with bases very close to 1 are not common.

But computers are getting faster 'exponentially'

Suppose

- This year's computer takes 1 ns per operation.
- Next year's computer is twice as fast as this year's.
- 1 day is a reasonable time to solve our problem.

For an algorithm that takes N^2operations

- This year: $N = 9.2 \times 10^6$.
- Next year: $N = 13 \times 10^6$.

For an algorithm that takes 2^N operations

$\log_2 \left(60 \times 60 \times 24 \times 10^9\right) = 46.296$

- This year: $N = 46$.
- Next year: $N = 47$.

An intractable algorithm

A **propositional formula** is a formula made using boolean variables (propositional variables), logical connectives, such as $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and parentheses.

A propositional formula is said to be **satisfiable** iff there is at least one assignment of boolean values to its variables so that it evaluates to true.

In other words, a propositional formula is **satisfiable** iff there is at least one row in its truth table so that it evaluates to true.

Examples:

v_0	v_1	$(v_0 \vee v_1) \wedge (\neg v_0 \wedge \neg v_1)$
false	false	false
false	true	false
true	false	false
true	true	false

v_0	v_1	$(v_0 \wedge v_1) \vee (\neg v_0 \wedge \neg v_1)$
false	false	true
false	true	false
true	false	false
true	true	true

Consider this problem.

- **Input**: A propositional formula ϕ of size N with propositional variables $\{v_0, v_1, ..., v_{m-1}\}$
- **Output**: 'yes' if there exist values of the variables to make the formula true. Otherwise 'no'

This is called "the propositional satisfiability problem" (PSAT).

Examples

- Input: $(v_0 \vee v_1) \wedge (\neg v_0 \wedge \neg v_1)$ Output: no
- Input: $(v_0 \wedge v_1) \vee (\neg v_0 \wedge \neg v_1)$ Output: yes. For example $\{\text{“}v_0\text{”} \mapsto \mathfrak{false}, \text{“}v_1\text{”} \mapsto \mathfrak{false}\}$

Backtracking algorithm

let $\{v_0, v_1, ..., v_{m-1}\}$ be the variables of ϕ
for $v_0 \in \{true, false\}$
  for $v_1 \in \{true, false\}$
    ...
      for $v_{m-1} \in \{true, false\}$
        if v satisfies ϕ output yes and stop
output no

We can also write it, in more Java-like fashion, with a recursive subroutine

```
public class Formula {
    public boolean evaluate( boolean[] v ) { ... }
    ⋮
```

```
private boolean search( int i, boolean[] v ) {
    // pre: v.length==getVarCount()
    if( i == v.length ) {
        return evaluate( v ) ; }
    else {
        for( int k=0 ; k < 2 ; ++k ) {
            v[i] = (k==0) ;
            if( search( i+1, v ) ) return true ; }
        return false ; } }

public boolean isSatisfiable() {
    boolean[] v = new boolean[getVarCount()] ;
    return search(0, v ) ; } }
```

Tractable and intractable problems

If any algorithm for a problem is tractable, the problem is *tractable*.

If every algorithm for a problem is intractable, the problem is *intractable*.

Decision problems are problems with yes/no outputs.

We call the set of tractable decision problems $\mathbf{P}$.

Some tractable problems:

- Sorting has an $O(N \log N)$ algorithm and so is tractable.
- Determining the shortest path between two points in a graph is tractable.
- Determining if an N digit number is prime is tractable.

It is in $\mathbf{P}$.

Intractable problems:

- Determining if a theorem has a proof in certain simple logics is intractable.

Is PSAT intractable?

The algorithms given above are intractable. They are $\Omega(2^m)$.

But that does not imply that PSAT is an intractable problem.

After all, couldn't there be better algorithms?

One bad algorithm does not imply the problem is hard!

The truth is that I don't know if PSAT is intractable.

- I don't know a fast way to solve the problem.
  - Can **you** find one?
- I can't show that all algorithms for PSAT are slow
  - Can **you** show there is no fast algortihm?

Some problems whose tractability is not known!

- Given a number, what are its factors.
- Find the longest (cycle free) path between two points in a graph.
- Find the shortest tour in a graph that visits every node. (TSP)
- Given a formula, is it satisfiable (PSAT).
- Given a map (planar graph), can it be coloured with 3 colours?
- Given a graph, how many colours are needed to colour it.
- Given a sequence of machine code with no branches, what is the shortest equivalent sequence.

- Given a sequence of C code with no branches, what is the minimum number of registers required to execute it.
- Given a set of files, what is the minimum number of disks (of fixed size) required to hold them all.

Amazingly, with the exception of factoring, the above problems all have similar tractability.

I.e. they are either all tractable or all intractable.

We just don't know which.

These problems form a class known as NP-equivalent.

We say P is *as tractable as* Q if 'Q is tractable' implies 'P is tractable'

We say P is *as intractable as* Q if 'Q is intractable' implies 'P is intractable'

We're going to spend the next while looking at the tools you need

- to show that problems are as tractable as the NP-equivalent one and
- to show that problems are as intractable as the NP-equivalent ones

If a problem is both as-tractable-as and as-intractable-as the NP-equivalent problems, then it too is NP-equivalent.

Decision problem versions

Most of the problems just mentioned are not decision problems.

However, one can usually find a related decision problem that is equally tractable.

A example: The travelling salesperson problem.

- The travelling salesperson problem (optimization version)
  - Input: A list of cities and, for each pair of cities, an integer distance.
  - Output: A shortest tour that visits every node.
- The travelling salesperson problem (decision version) (TSPD)
  - Input: A list of cities, for each pair of cities, an integer distance, and an integer k
  - Output: 'Yes' if there is a tour that visits every node and has length $\leq k$, otherwise, 'no'

The optimization version is intractable if the decision version is intractable.

Another example. Factoring

The input size is the number of bits needed to represent n.

- Factoring (search version)
  - Input: a natural number n
  - Output: the smallest factor of n greater than 1.
- Factoring (decision version)
  - Input: natural numbers n and k
  - Output: 'Yes' if n has a factor smaller than k. Otherwise, 'no'.

The search version is tractable iff the decision version is tractable. (Consider binary search.)

Because we can find decision problems that are just as hard as the search problems, we will focus on decision problems.

NP problems

Recall that $\mathbf{P}$ is the set of decision problems with polynomial time solutions.

Magic Coins

A *magically nondeterministic* algorithm for a decision problem is allowed one more operation, which we will assume is $\Theta(1)$ time.

- **boolean** magicCoin()

magicCoin returns true or false, but magically always returns an answer that leads to a "yes" output if there is one.

An algorithm for PSAT using magicCoin()

```
public class Formula {
    public boolean evaluate( boolean[] v ) { ... }
    ⋮
    public boolean isSatisfiable() {
        boolean[] v = new boolean[getVarCount()] ;
        for( int i=0 ; i < m ; ++i ) v[i] = magicCoin() ;
        return evaluate( v ) ; } }
```

This algorithm has the properties that

- the input formula is satisfiable iff there exists a sequence of answers from the magic coin that leads the algorithm to answer "Yes"
- it is a polynomial time algorithm

However, note that because this algorithm relies on magical nondeterminism — a mechanism that is, in general, impractical to implement— we do not consider that it is a regular polynomial time algorithm.

Thus, we do not consider this algorithm as evidence that $\mathrm{PSAT} \in \mathbf{P}$

NP (magic coin definition)

NP is the set of decision problems with magically nondeterministic, polynomial-time algorithms.

Since the above algorithm is a linear time algorithm, PSAT is in **NP**.

NP (ordinary coin definition)

Suppose we only have only an ordinary coin, i.e. a coin that unpredicatably sometimes gives heads and sometimes gives tails. It doesn't need to be a fair coin: one that comes up heads 60% of the time and tails %40 of the time, is ok.

NP is the set of decision problems for which we can write a polynomial time algorithm (using an ordinary coin) such that

- When the correct answer is 'no' the algorithm outputs 'no'.
- When the correct answer is 'yes', the algorithm *may* output 'yes'.

Equivalence with the magic coin definition

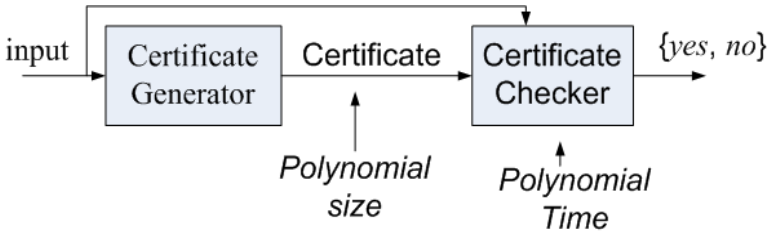
- If we have an ordinary-coin algorithm, we can replace the ordinary coin with a magic coin to get a magic-coin algorithm.
- If we have a magic-coin algorithm, we can replace the ordinary coin with an ordinary coin to get an ordinary-coin algorithm.

NP (certificate checking definition)

$\mathbf{NP}$ can also be defined as decision problems whose "yes" answers can be checked in polynomial time given some evidence of polynomial size.

Imagine an algorithm for a decision problem that has two parts.

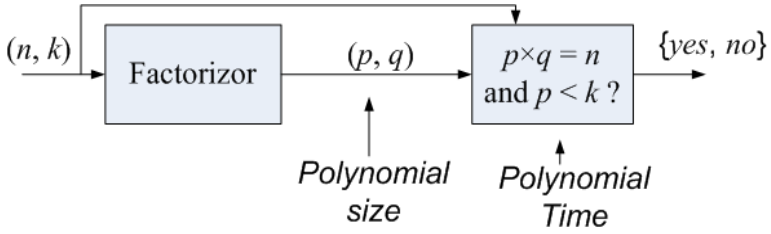
- The first part generates a "certificate" of polynomial size. The first part can take any amount of time.
- The second part reads the input and the "certificate" and computes the answer ('yes' or 'no') in polynomial time.



Define a problem to be in $\mathbf{NP}$ iff there is a (nonmagical) algorithm with this structure that solves the problem.

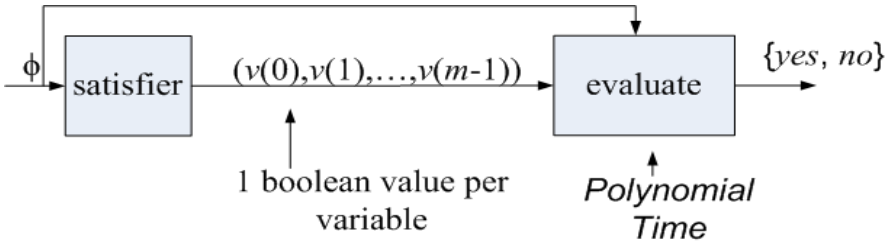
Example: We can construct an algorithm for factoring: Does n have a factor of size less than k?

- If n is has a factor smaller than k, the certificate generator outputs such a facto p and $q = n/p$.
- Otherwise, the certificate generator outputs any two numbers less than n.
- The certificate checker checks that $p < k$ and that $pq = n$.



Example: We can construct an algorithm for PSAT.

- If the input formula ϕ is satisfiable, first certificate generator outputs an assignment of boolean values to the propositional variables that satisfies ϕ.
- Otherwise the certificate generator outputs any assignment of boolean values to the propositional values.
- The certificate checker reads in the formula and the assignment and outputs the value of the formula for that assignment.



Problem	Certificate	Check
Is an N bit number composite?	2 proposed factors	Multiply and compare
PSAT	Values for the variables	Evaluate ϕ and compare to true
Can a map be 3 coloured?	A colour scheme	Compare colours of all adjacent countries
Is there a tour of a graph shorter than length k?	A tour	Calculate the length of the tour and compare to k.

Equivalence with the magic coin definition

- The sequence of magic coin results will serve as a certificate.
- The binary coding of a certificate can serve as a sequence of magic coin results.

$$\mathbf{P} \subseteq \mathbf{NP}$$

NP and co-NP (optional)

The complement $\overline{Q}$ of a decision problem Q is simply the problem that has 'no' answers and 'yes' answers the other way around. I.e.

$$\overline{Q}(x) = \neg Q(x), \text{ for all inputs } x$$

co-$\mathbf{NP}$ is the set of problems whose complements are in $\mathbf{NP}$.

For example $\overline{\text{PSAT}}$ (unsatisfiability) is the problem of determining whether a propositional formula is *unsatisfiable.* (I.e., it is false for every line of the truth table.)

$\mathbf{P} \subseteq \text{co-}\mathbf{NP}$ For any problem Q in $\mathbf{P}$, its complement $\overline{Q}$ will also be in $\mathbf{P}$ and thus also in $\mathbf{NP}$.

It is widely believed that $\mathbf{NP}$ and co-$\mathbf{NP}$ are not equal.

For example, it is widely beleived that PSAT is in $\mathbf{NP}$, but not in co-$\mathbf{NP}$ and (equivalently) that $\overline{\text{PSAT}}$ is in co-$\mathbf{NP}$, but not in $\mathbf{NP}$.

It might be helpful to see why $\overline{\text{PSAT}}$ can not be trivially proved to be in $\mathbf{NP}$.

Magic coin perspective

Consider the following **nonproof** that $\overline{\text{PSAT}}$ is in $\mathbf{NP}$.

Start of nonproof: Here is a Java algorithm for $\overline{\text{PSAT}}$ using magicCoin()

```
public class Formula {
    public boolean evaluate( boolean[] v ) { ... }
    ⋮
    public boolean isUnsatisfiable() {
        boolean[] v = new boolean[getVarCount()] ;
        for( int i=0 ; i < m ; ++i ) v[i] = magicCoin() ;
        return ! evaluate( v ) ; } } // Note the "!".
```

If the formula is unsatisfiable, the algorithm certainly returns true ('yes'). No magic is needed.

If the formula is satisfiable, the coin (magically) ensures that v is set to a line in the truth table for which the evalutation is true and thus the algorithm returns false ('no'). QED.

End of nonproof.

Why is this a nonproof? The magic coin is (by definition) helpful only for 'yes' inputs. For 'no' inputs, the coin is no better than any other coin. The nonproof is relying on the coin behaving magically for 'no' inputs.

Certificate perspective.

It is not obvious that there is always a sufficiently short certificate to show that a formula is unsatisfiable. Here are two examples of certificates that won't work.

- If the certificate consists of an assignment for which the formula is false, this is insufficient evidence, as there may be other assignments for which the formula is true.
- If the certificate consists of every row for which the formula is false, it may be more than polynomially long ($m \times 2^m$ bits) and will take too long to check.

(End of optional section.)

Magic nondeterminsm vs regular nondeter-

minism (optional)

[This section is entirely optional. It is here for the gratification of any student who might be wondering how the nonderminism of the magic coins used here (and in the earlier section on NDFRs) relates to the nondeterminism we saw in the first part of the course.]

Let's look at magic nondeterminism from the point of view of nondeterministic specifications as presented earlier in the course.

Define a programming construct

$$\mathbf{try}\ f\ \mathbf{else}\ g$$

This is defined by

$$(\mathbf{try}\ f\ \mathbf{else}\ g)\,(i \dagger o) = \begin{array}{l}((\exists \dot{o} \cdot f\,(i \dagger \dot{o})) \wedge f\,(i \dagger o)) \\ \vee\ (\neg\,(\exists \dot{o} \cdot f\,(i \dagger \dot{o})) \wedge g\,(i \dagger o))\end{array}$$

Essentially $\mathbf{try}\ f\ \mathbf{else}\ g$ behaves like f, if possible, and otherwise behaves like g.

Note that if f is implementable then $\mathbf{try}\ f\ \mathbf{else}\ g$ is the same as f. The construct is only interesting if f is not implementable.

Note that $\mathbf{try}\ f\ \mathbf{else}\ g$ is implementable if g is implementable, even if f is not implementable.

Now define a construct that is not always implementable:

$$\mathbf{force}\ \mathcal{A}$$

ensures that $\mathcal{A}$ is true, but does not change any variables

$$(\mathbf{force}\ \mathcal{A}\ ) = \langle \mathcal{A} \rangle \wedge \mathbf{skip}$$

Force is generally not implementable, as it restricts its initial state to one where $\mathcal{A}$ is true.

A command $f; \mathbf{force}\ \mathcal{A}$ accepts all behaviors accepted by f such that $\mathcal{A}$ is true of the final state.

Finally we define an ordinary (nonmagical) nondeterministic coin flip as follows.

$$\mathbf{flip}\ \mathcal{V} = ((\mathcal{V} := \mathfrak{true}) \lor (\mathcal{V} := \mathfrak{false}))$$

Now suppose the result of our algorithm is to go into a boolean variable b. For example, the specification may be $\langle b' = Q(x) \rangle$ where x is the input and Q is a boolean function.

We can understand a polynomial time 'magic coins algorithm' as an algorithm of the following form

$$\mathbf{try}\ (f; \mathbf{force}\ b)\ \mathbf{else}\ b := \mathfrak{false}$$

where f is a polynomial-time algorithm that uses no nondeterminism other than $\mathbf{flip}$ commands and b is the boolean variable that receives the decision. The specification for f here is twofold:

- In the case of a 'no' input, f must ensure b' is false $\langle \neg Q(x) \Rightarrow \neg b' \rangle \sqsubseteq f$ That is
$$\langle b' \Rightarrow Q(x) \rangle \sqsubseteq f$$
- In the case of a 'yes' input, f must allow the possibility that b' is true:
$$\forall i \mid Q\,(i\,(\text{“}x\text{”})) \cdot \exists o \mid o\,(\text{“}b\text{”}) \cdot f(i \dagger o)$$

Now one can show that

$$\langle b' = Q(x) \rangle \sqsubseteq \mathbf{try}\ (f; \mathbf{force}\ b)\ \mathbf{else}\ b := \mathfrak{false}$$

For example we can refine $\langle b' = PSAT(\phi) \rangle$ by

$$\begin{array}{l} \\ \textbf{try} \\ \\ \textbf{else } b := \mathfrak{false} \end{array} \left(\begin{array}{l} \textbf{let } m = \text{the number of variables in } \phi \cdot \\ \textbf{var } v : \mathbb{B}^m \cdot \\ \textbf{for } i \in \{0, ..m\} \textbf{ do flip } v[i] \ ; \\ b := \text{evaluate}(\phi, m) \ ; \\ \textbf{force } b \end{array} \right)$$

The effect of the $\textbf{force } b$ is to ensure that the $\textbf{flip}$s go the right way, if at all possible. In a sense, it adds the magic to the nondeterministic flips.

(End of optional section.)

NP-hard and NP-easy

The following definitions apply to all problems, not just decision problems.

A problem P is called $\mathbf{NP}$-*easy* if it is as tractable as some problem in $\mathbf{NP}$

A problem P is called $\mathbf{NP}$-*hard* if it is as intractable as every problem in $\mathbf{NP}$

A problem P is called $\mathbf{NP}$-*equivalent* if it is both $\mathbf{NP}$-hard and $\mathbf{NP}$-easy.

Next we look at a technique to show that one decision problem is as-tractable-as another.

Reducibility

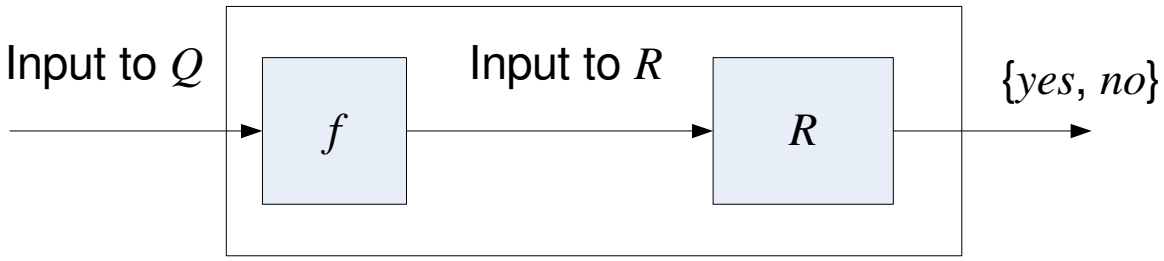
We can *polynomially reduce* a decision problem Q to a decision problem R, if we can find an algorithm of the following form for Q that returns the correct decision:

Read input x for problem Q
Transform x to an input y for problem R in polynomial time
Output $R(y)$

Definition: A decision problem Q *can be (polynomially) reduced to* a decision problem R iff there exists a function f, that has a polynomial time algorithm, such that

$$Q(x) = R(f(x)), \text{ for all } x$$

Notation: We write $Q \xrightarrow{\text{p}} R$ to mean Q can be polynomially reduced to R.



If Q can be reduced to R ($Q \xrightarrow{\text{p}} R$) then:

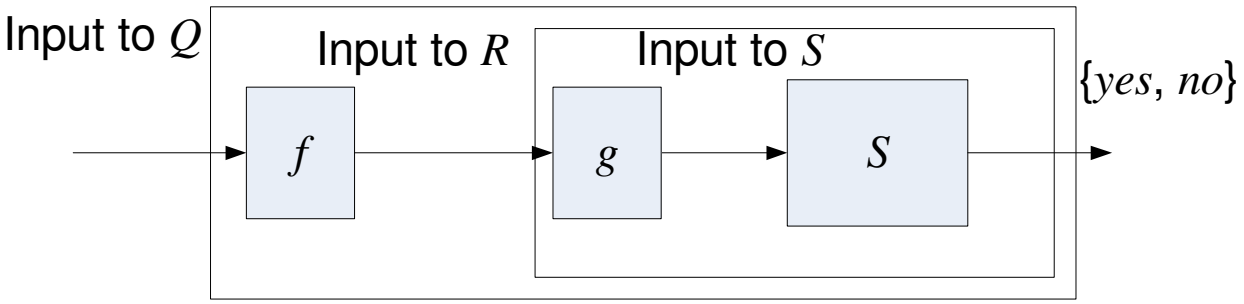
- If R is tractable
  - then Q is tractable.
- If Q is intractable,
  - then R is intractable.

Now if R is reducible to Q and Q is reducible to R (i.e. $Q \xrightarrow{\text{p}} R \wedge R \xrightarrow{\text{p}} Q$) then

- R is tractable iff Q is tractable
- R is intractable iff Q is intractable

If Q can be reduced to R and R can be reduced to S (i.e. $Q \xrightarrow{\text{p}} R \wedge R \xrightarrow{\text{p}} S$) then

- Q can be reduced to S.(i.e. $Q \xrightarrow{\text{p}} S$)
- Proof: Suppose $Q(x) = R(f(x))$, for all x, and $R(x) = S(g(x))$, for all x, where f and g take polynomial time.
  - Let $h(x) = g(f(x))$, for all x;. h will take polynomial time
  - $Q(x) = R(f(x)) = S(g(f(x))) = S(h(x))$, for all x.



So, if we have a cycle of decision problems Q_0, Q_1, ..., Q_n, with $Q_n = Q_0$, and each is reducible to the next

- then either all are tractable or none are tractable.

An easy example

An *undirected graph* is a pair of sets (V, E) where $E \subseteq \{a, b \in V \mid a \neq b \cdot \{a, b\}\}$.

- The set V is called the set of 'nodes' (or 'vertices')
- The set E is called the set of 'edges' (or 'arcs')

A *path* in a graph is a sequence of nodes $[a_0, a_1, .., a_k]$ where each $\{a_i, a_{i+1}\}$ is in E.

A *cycle* (or *circuit*) is a path where $\{a_k, a_0\}$ is also an edge.

A *Hamiltonian circuit* is a cycle that contains every node once.

Hamiltonian circuit problem (HC):

- INPUT: An undirected graph $G = (V, E)$
- QUESTION: Is there a Hamiltonian circuit?

Traveling salesperson decision problem (TSDP):

- INPUT: A finite set of cities C, a distance function $d : C \times C \to \mathbb{N}$, and an integer bound B.
- QUESTION: Is there a tour that visits every city once and then returns to the first city, of total length $\leq B$? I.e. is there way to number all the cities $\left[c_0, c_1, ..., c_{|C|-1}\right]$ such that

$$d(c_{|C|-1}, c_0) + \sum_{i \in \{0,..|C|-1\}} d(c_i, c_{i+1}) \leq B$$

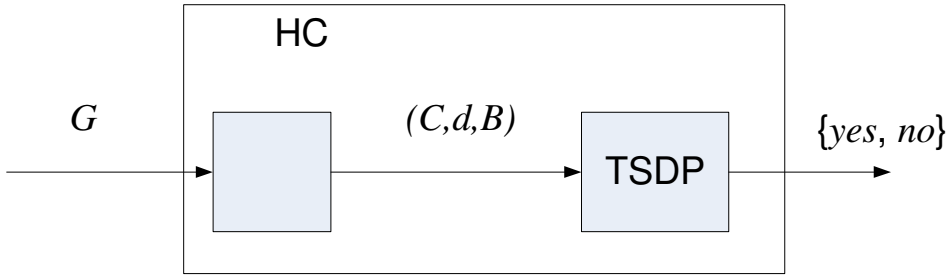
Finding a transformation

We will show that the Hamiltonian circuit problem is reducible to the TSDP. And thus that the Hamiltonian circuit problem is no more intractable than the TSDP.

(i) **Transformation**: Given an input $G = (V, E)$ for the HC problem, construct an input (C, d, B) for TSDP as follows:

Let $C = V$, let $B = |V|$ and let

$$d(a, b) = \begin{cases} 1 \text{ if } \{a, b\} \in E \\ 2 \text{ if } \{a, b\} \notin E \end{cases}$$



We need to argue

(ii) that this transformation takes only polynomial time, and

(iii) that the answer to TSDP on (C, d, B) is yes if and only if the answer to HC on G is yes, where G is any input for HC and (C, d, B) is the result of the transformation applied to G.

(ii) The transformation is clearly linear time assuming a reasonable representation of the graph.

(iii) Let G be any graph at all and let (C, d, B) be the output of the transformation if G is the input.

(α) We must show that: If G is a 'yes' input for HC, then (C, d, B) is a 'yes' input for TSDP.

If HC gives 'yes' on G, then there is a cycle $\left[a_0, a_1, ..., a_{|V|-1}\right]$ containing every node once. Thus, $\{a_0, a_1\}$, $\{a_1, a_2\}$, etc, and $\left\{a_{|V|-1}, a_0\right\}$ are all the edges in G. Because of the way d is constructed, $d(a_0, a_1) = 1$, $d(a_1, a_2) = 1$, etc, and $d(a_{|V|-1}, a_0) = 1$, so this order gives a tour of the cities of length

$$d(a_{|V|-1}, a_0) + \sum_{i \in \{0,..|C|-1\}} d(a_i, a_{i+1}) = |V| = B$$

(β) We must show that: If (C, d, B) is a 'yes' input for TSDP, then G is a 'yes' input for HC.

If TSDP gives 'yes' on (C, d, B) then there is a tour $\left[a_0, a_1, ..., a_{|V|-1}\right]$ of length $|V|$. Thus

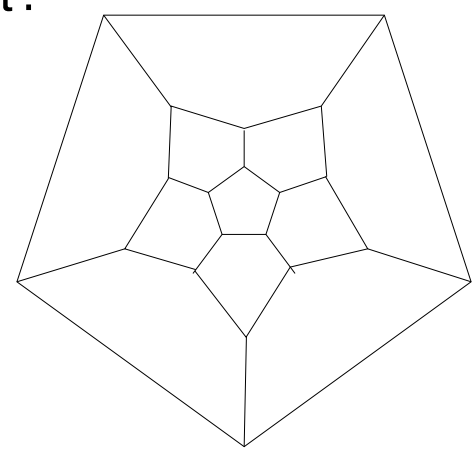
$$d(a_{|V|-1}, a_0) + \sum_{i \in \{0,..|C|-1\}} d(a_i, a_{i+1}) = |V|$$

Since there are $|V|$ terms in

$$d(a_{|V|-1}, a_0) + \sum_{i \in \{0,..|C|-1\}} d(a_i, a_{i+1})$$

and each term is 1 or 2, each term must be 1. Thus. from the construction of d, we have that $\{a_0, a_1\}$, $\{a_1, a_2\}$, etc, and $\left\{a_{|V|-1}, a_0\right\}$ are all the edges in G and so $\left[a_0, a_1, ..., a_{|V|-1}\right]$ is a Hamiltonian circuit.

By the way, 19th c. mathematician William Rowan Hamilton hoped to get rich selling a pentagonal dodecahedron with the challenge to find a cyclic path along the edges that reached each node. Can you find a Hamiltonian circuit?



See

http://www.puzzlemuseum.com/month/picm02/200207icosian.htm

Cook's theorem

A conjunctive normal form (CNF) propositional formula is of the form

$$c_0 \wedge c_1 \wedge \ldots \wedge c_n$$

where each c_i is of the form

$$(a_{i,0} \vee a_{i,1} \vee \ldots \vee a_{i,k_i})$$

where each $a_{i,j}$ is either a variable or the negation of a variable.

Example

$$(v_0 \vee v_1) \wedge (\neg v_0 \vee v_2 \vee \neg v_3) \wedge (v_0 \vee \neg v1 \vee \neg v_2)$$

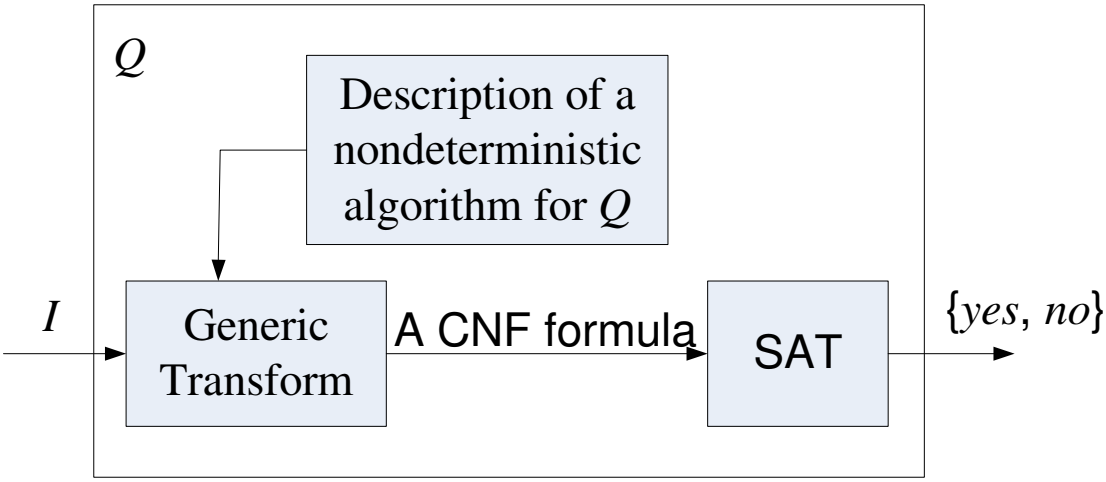
SAT is the problem of determining whether such a formula is satisfiable

Cook's theorem (Stephen Cook, 1971)

*Every problem in **NP** is reducible to SAT*

Proof: The basic idea is.

- For any problem Q in $\mathbf{NP}$, there is (by definition) a polynomial-time magically-nondeterministic algorithm for it.
- For each input I, of size N, the algorithm will take no more than $p(N)$ steps where p is a polynomial.
- For each input I we can systematically construct (from the algorithm and the number of steps it takes) a CNF formula of polynomial size that is satisfiable iff the algorithm results in a 'yes'.



- This gives a polynomial time reduction from Q to SAT.

Wikipedia gives more detail. See http://en.wikipedia.org/wiki/Cook%27s_theorem .

NP-Completeness

Definition: We say that a problem R is *NP-complete* ($\mathbf{NPC}$) iff it is in $\mathbf{NP}$ and every problem in $\mathbf{NP}$ is reducible to R.

(Equivallenty a problem is $\mathbf{NP}$-complete iff it is in $\mathbf{NP}$ and is $\mathbf{NP}$-hard.)

Thus Cook's theorem says that SAT is $\mathbf{NP}$-complete.

To say a problem R is $\mathbf{NP}$-complete is to say

- if R is tractable then every problem in $\mathbf{NP}$ is tactable.

And so any $\mathbf{NPC}$ problem is a *least tractable* problem in $\mathbf{NP}$.

Summary: the following are equivalent statements

- $\mathbf{P} = \mathbf{NP}$
- all $\mathbf{NPC}$ problems are tractable
- at least one $\mathbf{NPC}$ problem is tractable

And the following are equivalent statements

- $\mathbf{P} \neq \mathbf{NP}$
- at least one $\mathbf{NPC}$ problem is intractable
- all $\mathbf{NPC}$ problems are intractable

Implications:

- We should look for a polynomial time solution to some $\mathbf{NPC}$ problem. This would prove $\mathbf{P} = \mathbf{NP}$ and immediately yield polynomial time algorithms for all problems in $\mathbf{NP}$, and
- We should try to find a nonpolynomial time lower-bound for some $\mathbf{NPC}$ problem, thus proving $\mathbf{P} \neq \mathbf{NP}$.

People have been trying to resolve this question since 1972 with no luck yet.

The Clay Mathematics Institute has included this question among 7 problems for which it is willing to offer a $1 million prize. See

http://www.claymath.org/millennium/P_vs_NP/

A very practical consequence is that

- Any problem that is in $\mathbf{NPC}$ will not have a polynomial time algorithm unless $\mathbf{P} = \mathbf{NP}$.

Thus if you can show a problem is in $\mathbf{NPC}$ then, *even if it is a new problem*, you know that over 30 years of research has failed to find a poly-time algorithm a large number of equivalently hard problems.

So how do we show that a problem is NP-Complete?

Theorem: If Q is in $\mathbf{NP}$, R is $\mathbf{NP}$-Complete, and R can be reduced to Q then Q is also **NP**-complete.

Proof:

- Recell that Q is NP-Complete iff every problem in $\mathbf{NP}$ can be reduced to Q.
- Assume Q is in $\mathbf{NP}$, R is $\mathbf{NP}$-Complete, and R can be reduced to Q
- We must show that any problem S in $\mathbf{NP}$ can be reduced to Q.
  - Since R can be reduced to Q, there is a polynomial time f such that $R(y) = Q(f(y))$, for all y
  - Consider any problem S in $\mathbf{NP}$.
  - Since R is $\mathbf{NP}$-Complete, there is a polynomial time g such that $S(x) = R(g(x))$, for all x
  - Now $S(x) = Q(f(g(x))$, for all x.
  - Since f and g are polynomial, so is their composition. Thus S can be reduced to Q.

More $\mathbf{NP}$-Complete problems.

Using the last theorem and Cook's theorem, we can prove lots of useful problems are $\mathbf{NP}$-Complete.

Example: PSAT

PSAT is in $\mathbf{NPC}$ since we can transform any input for SAT to an input for PSAT by doing nothing.

Example: 3SAT

The problem 3SAT is the same as SAT, but each disjunction must have exactly 3 disjuncts:
$(a_{0,0} \vee a_{0,1} \vee a_{0,2}) \wedge (a_{1,0} \vee a_{1,1} \vee a_{1,2}) \wedge \ldots \wedge (a_{k,0} \vee a_{k,1} \vee a_{k,2})$
We can transform any input for SAT to an input for 3SAT (how?)

Example: Vertex cover

We can transform any input for 3SAT to an input for the vertex cover problem

We'll do this later.

Example: Hamiltonian circuit

We can transform any input for the vertex cover problem to an input for the HC problem.

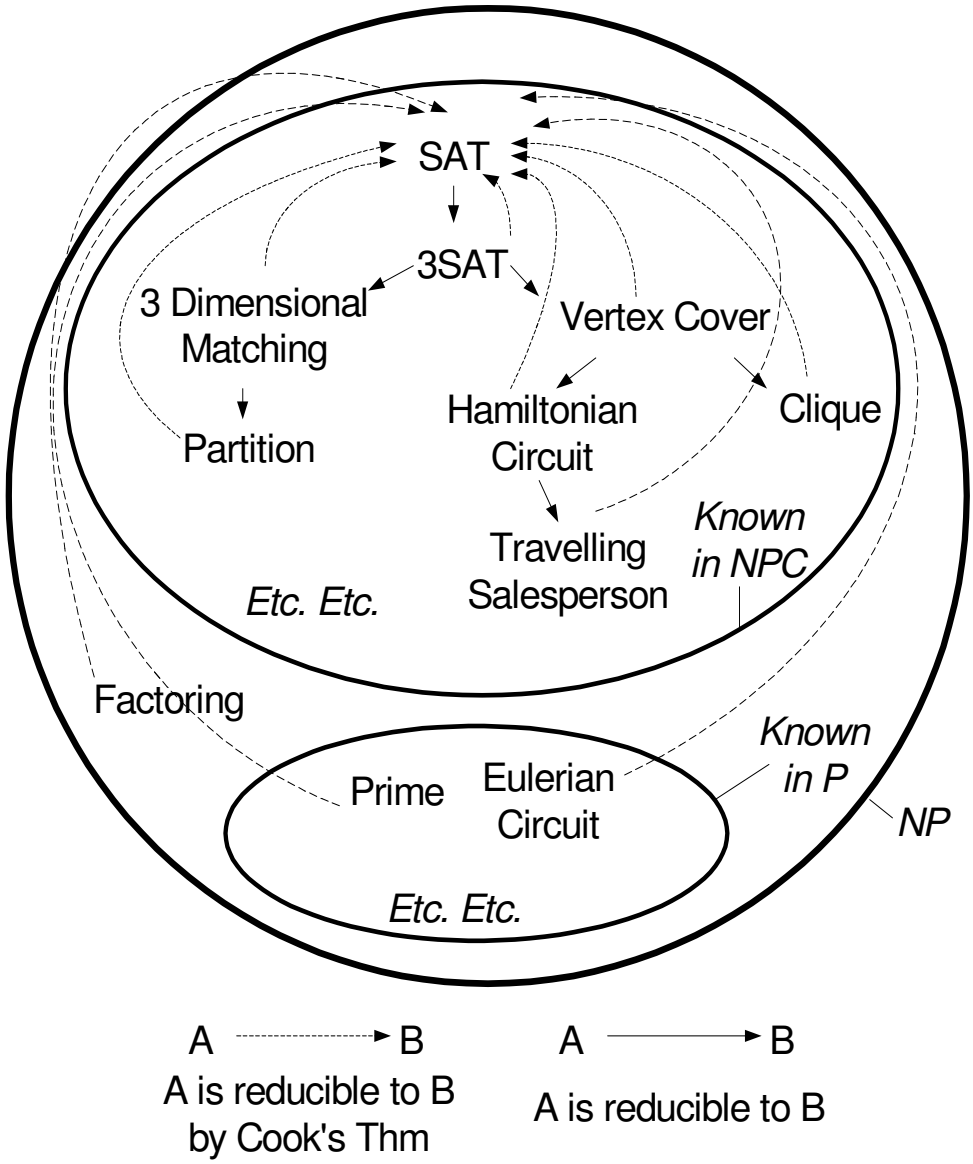
Non-Example: Factoring

Given an N bit number i and a number k, does i have a factor less than k? Certainly this is in $\mathbf{NP}$, but it is not known to be in $\mathbf{NPC}$.

Non-Example: Prime

Given an N bit number i, is i prime? Recently proved in $\mathbf{P}$.

In a picture



Some problems in NP