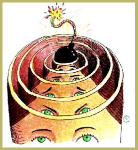
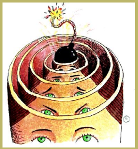


Lecture 32: **P** and **NP** Problems

Aims:

- To describe two classes of decision problems: **P** and **NP**;
- To pose the million dollar question: does **P** = **NP**?



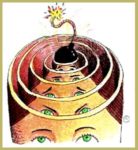
32.1. Introduction

32.1.1. Recap

- For some problems there are known polynomial-time algorithms;
- For others, we have proofs that no polynomial-time algorithm can exist;
- But for others, we have only exponential-time algorithms but no proof that no polynomial-time algorithm can exist.
- Despite the holes in our knowledge, Computer Scientists have made some progress.
- Computer Scientists have been able to find relationships between some of the third kind of problem.
  - E.g. They have been able to show that some problems are, in some sense, equivalent in complexity. (They do this using reductions.)
  - This isn't as good as closing the gap for these problems.
  - But it is significant because all these problems will stand or fall together, which makes them particularly worthy of further research effort.
  - We'll look at some of the results of this research.

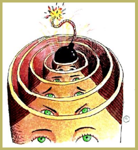
32.1.2. Decision Problems Again

- Researchers who work in Complexity Theory often concentrate on decision problems.
- Reminder: a decision problem is one whose return values are either YES or NO (or **true** or **false**, or 0 or 1).



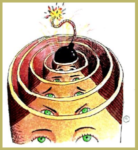
- Why concentrate on these?
  - If a problem has an exponential amount of output, then no amount of cleverness is going to find an algorithm that is polynomial. E.g. Towers of Hanoi, finding all permutations of n numbers, enumerating all Hamiltonian cycles.
  - For many non-decision problems, there are related decision problems. E.g. the TSP Search and Decision Problems. A non-decision problem (such as the TSP Search Problem) can often be turned into a decision problem by introducing a parameter k and asking if there is an answer that costs at least or at most k. If you can't find a polynomial-time algorithm for a decision problem, then you're not likely to find one for the non-decision problems to which it is related.

- Suppose you've a decision problem and an algorithm A that solves that decision problem.
- We'll say that algorithm A *accepts* input x if it returns YES.
- We'll say that algorithm A *rejects* input x if it returns NO.



32.2. The Complexity Class P

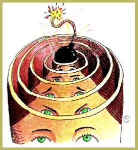
- *Definition:* The complexity class **P** is the set of all decision problems that can be solved with worst-case polynomial time-complexity.
- In other words, a problem is in the class **P** if it is a decision problem and there exists an algorithm that solves any instance of size n in $O(n^k)$ time, for some integer k. (Strictly, n must be the number of *bits* needed for a ‘reasonable’ encoding of the input. But we won’t get bogged down in such fine details.)
- So **P** is just the set of tractable decision problems: the decision problems for which we have polynomial-time algorithms.



32.3. The Complexity Class NP

- The second class of decision problems that we look at is called **NP**, which stands for *non-deterministically polynomial.*
- The definition of **NP** involves the idea of a *non-deterministic algorithm.*
- Suppose we introduce an extra primitive operation into DECAFF and we call the new language ND-DECAFF:
  - *choose*(m, n): this operation chooses an integer between m and n inclusive in a non-deterministic way

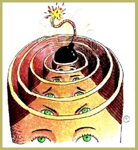
  But this does not work like a random number generator. It's more like tossing a magical coin or rolling a magical dice!
- The *choose* operation possesses magical insight: it always selects the possibility that leads to a YES answer, if the problem instance has a YES answer.
- If the instance has a NO answer, *choose* returns an arbitrary number between m and n inclusive.
- An algorithm that uses the *choose* operation zero, one or more times is referred to as a *non-deterministic algorithm.* It follows that normal (deterministic) DECAFF algorithms are trivial examples of ND-DECAFF algorithms. They use the *choose* operator zero times.
- An algorithm A *non-deterministically accepts* an input x if there exists a sequence of outcomes to the *choose* operation that A could make on input x such that A returns YES.
- *Definition:* The complexity class **NP** is the set of all decision problems that can be non-deterministically accepted in worst-case polynomial time.



- Notice that the definition of **NP** says nothing about the running time of producing NO answers.
- Of course, there's no computer than can toss magical coins or roll magical dice.
  - No conventional computer can carry out the *choose* operation.
  - No one has yet shown how even an unconventional computer (e.g. a quantum computer) could simulate a non-deterministic polynomial-time algorithm in polynomial time.
- So non-deterministic algorithms appear to be useless (we can't code them up and run them anywhere).
- The class **NP** may therefore seem pretty weird and pointless. Bare with it. It's a theoretical technicality in part of a broader argument that we will develop in the next lecture: it's part of a way of showing that some problems are related and so, if we ever find something out about the complexity of one of them, we may learn something about the complexity of all of them.

Class Exercise

$\mathbf{P} \subseteq \mathbf{NP}$. Why?



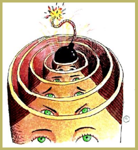
32.4. Proving that a problem is in NP

- Let's show that the Hamiltonian Cycle Decision Problem is in **NP**.
- We'll come up with a non-deterministic algorithm
  - It will use *choose* to guess, in polynomial-time, a possible cycle.
  - Then, it will take polynomial-time to check whether this possible cycle is, in fact, a cycle.
- Assume we have a graph $G = \langle V, E \rangle$. There are n vertices and they are numbered from 1 to n.

  Guess a possible cycle:

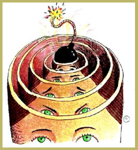
```
create an integer array $a[1 \ldots n]$;
for $i := 1$ upto $n$
{    $a[i] := choose(1, n)$;
}
```

What was this doing? It used *choose* to select n vertices and put them in an array. Remember, *choose* possesses magical insight. So, if there is a Hamiltonian cycle, it will magically pick n vertices that do form a Hamiltonian cycle. And, if there isn't a Hamiltonian cycle, it just pick n vertices arbitrarily. So now we need to check whether the array does or does not contain a Hamiltonian cycle.



Check the possible cycle:

```
// Create an array that will keep track of which vertices we have visited.
create a Boolean array $b[1 \ldots n]$;
// Record the fact that we have visited the vertex stored in $a[1]$.
$b[a[1]] :=$ **true**;
// Now visit each vertex stored in $a$ in turn.
**for** $j := 2$ **upto** $n$
{   If we have visited the vertex stored in $a[j]$ before...
    **if** $b[a[j]] =$ **true**
    {   **return** NO;
    }
    If there is no edge between $a[j-1]$ and $a[j]$...
    **if** $\{a[j-1], a[j]\} \notin E$
    {   **return** NO;
    }
    Record the fact that we have visited the vertex stored in $a[j]$.
    $b[a[j]] :=$ **true**;
}
We've nearly completed the cycle!
If there is no edge from $a[n]$ back to $a[1]$...
**if** $\{a[n], a[1]\} \notin E$
{   **return** NO;
}
**else**
{   **return** YES;
}
```

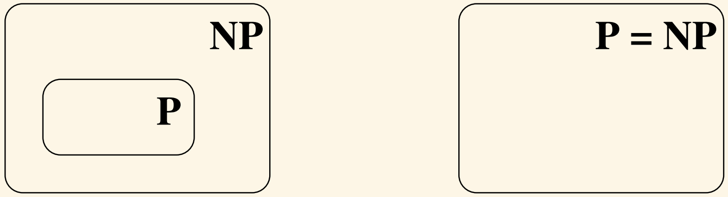


What was this doing? It uses array b to make sure that every vertex is used exactly once. It also checks that two consecutive vertices in a, $a[j-1]$ and $a[j]$ do, in fact, have an edge between them in the graph. And, finally, it makes sure that there is an edge between the last and first vertices in a.

- Both parts of the algorithm (the choosing and the checking) take polynomial time.
- This shows that the Hamiltonian Cycle Decision Problem is in **NP**.
- Note, when writing ND-DECAFF algorithms to confirm a problem is in **NP**, it is common for the algorithm to come in two parts like this.

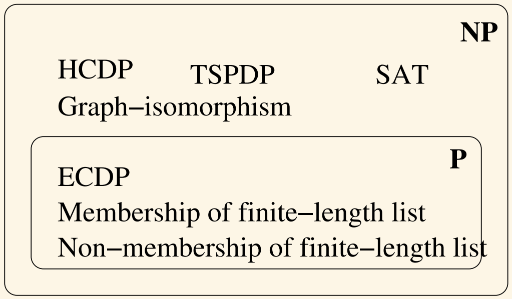
32.5. The P = NP Question

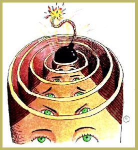
- We know that $\mathbf{P} \subseteq \mathbf{NP}$.
- But much more than that we don't know.
- The definition of **NP** allows for the inclusion of problems that may not be in **P**.
- But it may turn out that there are no such problems and that $\mathbf{P} = \mathbf{NP}$.



We don't know which of these is the case. We know $\mathbf{P} \subseteq \mathbf{NP}$. But we don't know whether $\mathbf{P} \subset \mathbf{NP}$ (left-hand diagram) or $\mathbf{P} = \mathbf{NP}$ (right-hand diagram).

- Let's put some actual problems into our diagram:


NP
HCDP
TSPDP
SAT
Graph–isomorphism
P
ECDP
Membership of finite–length list
Non–membership of finite–length list



- Key

EC/HC/TSP + DP:	Eulerian Cycle/Hamiltonian Cycle/Trav. Salesperson Decision Problem
SAT:	explained in the next lecture
Graph-isomorphism:	(roughly) are two graphs structurally equivalent?
Membership of finite-length list:	return YES if x is in list L; NO otherwise
Non-membership of finite-length list:	return YES if x is not in list L; NO otherwise

- The problems in the picture that are in **NP** but not in **P** are ones that we're not sure about:
  - there is no known polynomial-time algorithm;
  - but no proof of intractability.

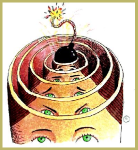
  (All we've managed to do is to show is that they're in **NP**.)

- So does $\mathbf{P} = \mathbf{NP}$? Or is $\mathbf{P} \neq \mathbf{NP}$ (i.e. $\mathbf{P} \subset \mathbf{NP}$?).
- If you can resolve this issue, you can win a million dollars.
  `http://www.claymath.org/Millennium_Prize_Problems/`

Acknowledgements:

The idea of explaining *choose* in terms of tossing a magical coin comes 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.