Hamiltonian Paths in Directed Graphs

A *Hamiltonian path* in a directed graph is a directed path that goes through each node exactly once.

The decision problem: *Given a directed graph G and nodes s and t in this graph, is there a Hamiltonian path from s to t in G?*

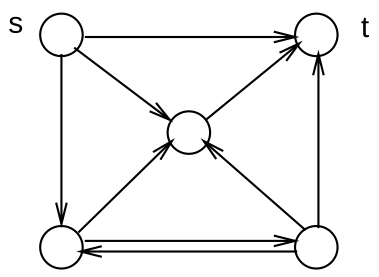
The corresponding language:

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph having a Hamiltonian path from } s \text{ to } t\}$$

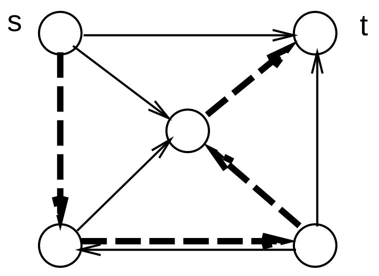
Brute-force (deterministic) decider for *HAMPATH*: Given directed graph G having m nodes, try every sequence of $m-2$ (or, if $s = t$, $m-1$) distinct nodes $r_1, r_2, \ldots, r_k$, all different from s and t, and test whether $(s, r_1), (r_1, r_2), \ldots, (r_k, t)$ are all directed edges in G.

There are $(m-2)!$ sequences of $m-2$ distinct nodes, so this is clearly not a polytime decider for *HAMPATH*.

Example: Does this directed graph have a Hamiltonian path from s to t?



Yes:



Hamiltonian Paths (Continued)

The decision problem: *Given a directed graph G and nodes s and t in this graph, is there a Hamiltonian path from s to t in G?*

The corresponding language:

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph having a Hamiltonian path from } s \text{ to } t\}$$

Theorem. $HAMPATH \in$ NP.

Proof. Here is a nondeterministic polytime decider for $HAMPATH$:

```
$N_{HAMPATH}$ =  "On input $\langle G, s, t\rangle$, where $G$ is a directed graph with nodes $s$ and $t$:
    1.   Nondeterministically create a permutation $r_1, r_2, \ldots, r_k$ of all nodes
         in $G$ not in $\{s, t\}$.
    2.   Check that $(s, r_1), (r_1, r_2), \ldots, (r_k, t)$ are all directed edges in $G$.
         If they all are, accept; otherwise, reject."
```

One way to implement stage 1, using a two-tape NTM:

- First mark s and t.
- Then repeatedly select an unmarked node nondeterministically and mark it and copy it to tape 2.
- Repeat this process until there are no unmarked nodes. At this point tape 2 contains the nondeterministically re-ordered list of nodes $r_1, r_2, \ldots, r_k$.

Correctness of $N_{HAMPATH}$: If there is a Hamiltonian path from s to t, the branch of the computation tree generating a corresponding ordering of nodes for this path will accept on such a branch. If there is no Hamiltonian path then no ordering will work and all computation branches will reject.

Time complexity of $N_{HAMPATH}$:

- Stage 1 can clearly be implemented in (nondeterministic) time polynomial in the number of nodes of G.
- Stage 2 also can clearly be implemented in time polynomial in the number of edges, which is at worst quadratic in the number of nodes.

Therefore $N_{HAMPATH}$ is a nondeterministic polytime decider for $HAMPATH$, which proves that $HAMPATH \in$ NP.

Polynomial Time Verification

Note that the nondeterministic decider for *HAMPATH* really just consists of two stages: a stage that nondeterministically "guesses" an answer, followed by a deterministic stage that verifies the correctness of the guess.

Definition. A *verifier* for a language L is a (deterministic) algorithm V where

$$L = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

The string c is called a *certificate* or *proof* of membership in L; it represents extra information that the verifier can use to decide whether w is a member in L.

The running time of a verifier is measured only in terms of the length of w, not the length of $\langle w, c\rangle$. A *polynomial time (polytime) verifier* is a verifier that runs in time polynomial in the length of w. A language L is *polynomially verifiable* if it has a polytime verifier.

For the languages we're dealing with that correspond to questions of the form *Does this problem have a solution?*, the certificate is typically a such a solution (however obtained), if any exists. Being able to answer the yes/no question given a solution is then simply a matter of verifying that the solution is valid (or not).

Alternative Characterization of NP

Theorem. $L \in$ NP iff L is polynomially verifiable.

Proof. First suppose there is an NTM N that decides L in (nondeterministic) polynomial time. Consider the following TM:

```
V = "On input $\langle w, c \rangle$, where $w$ and $c$ are strings:
      1.  Simulate $N$ on input $w$, but interpret every symbol in $c$ as a particular direction to go
          in $N$'s nondeterministic computation tree.
      2.  If this branch of $N$'s computation accepts, accept; otherwise, reject."
```

In stage 1 the string c is interpreted as the contents of the address tape as used in the 3-tape deterministic TM simulation of an NTM.

This clearly runs in polynomial time. Also, if there is some accepting branch of N's computation tree when presented with input w, then there is a certificate c corresponding to that branch and V will then accept $\langle w, c \rangle$. If N rejects w, then all branches reject, so V will not accept $\langle w, c \rangle$ for any c.

Now suppose there is a polytime verifier V for L that runs in time n^k. Consider the following NTM:

```
N = "On input $w$ of length $n$:
      1.  Nondeterministically select a string $c$ of length at most $n^k$.
      2.  Run $V$ on input $\langle w, c \rangle$. If it accepts, accept; otherwise, reject."
```

This NTM accepts w iff there is a certificate c such the verifier V accepts $\langle w, c \rangle$. Stage 2 clearly runs in polynomial time. To see that stage 1 can be implemented to run in polynomial time, consider the process of generating each symbol in c. This can clearly be done in polynomial time. Even if generating the various symbols (say, nearer the beginning compared to nearer the end) takes different amounts of time, as long as each is polynomial in n, the overall time is the sum of n^k polynomials, which is itself a polynomial. Therefore this NTM is a nondeterministic polytime decider for L.

HAMPATH Revisited

The decision problem: *Given a directed graph G and nodes s and t in this graph, is there a Hamiltonian path from s to t in G?*

The corresponding language:

$$\mathit{HAMPATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph having a Hamiltonian path from } s \text{ to } t\}$$

Theorem. *HAMPATH* $\in$ NP.

Proof. This time we use a polytime verifier. First, let the certificate c be a list $r_1, r_2, \ldots, r_k$ of all nodes in G not in $\{s, t\}$, where $k = m - 2$ (or $m - 1$ if $s = t$) and m is the number of nodes in G.

```
V_HAMPATH =  "On input $\langle\langle G, s, t\rangle, c\rangle$:
             1.   Check that $(s, r_1), (r_1, r_2), \ldots, (r_k, t)$ are all directed edges in $G$.
                  If they all are, accept; otherwise, reject."
```

Note that this is just the second stage of the NTM $N_{HAMPATH}$ that we used to prove this theorem earlier.

We already showed that this runs in time polynomial in the size of G, so it's a polytime verifier for *HAMPATH*, giving an alternative proof that *HAMPATH* $\in$ NP.

Hamiltonian Paths in Undirected Graphs

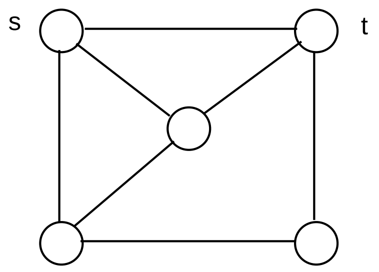
A *Hamiltonian path* in a undirected graph is, of course, a path (now undirected) that goes through each node exactly once.

The decision problem: *Given an undirected graph G and nodes s and t in this graph, is there a Hamiltonian path from s to t in G?*

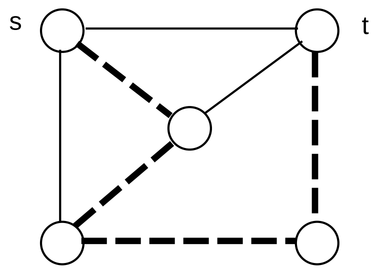
The corresponding language:

$$\mathit{UHAMPATH} = \{\langle G, s, t\rangle \mid G \text{ is a undirected graph having a Hamiltonian path from } s \text{ to } t\}$$

Example: Does this undirected graph have a Hamiltonian path from s to t?



Yes:



Theorem. *UHAMPATH* $\in$ NP.

Proof. Clearly membership in this language can be verified in polynomial time given a certificate consisting of a list of all nodes different from s and t, just as with *HAMPATH*.

Cliques in an Undirected Graph

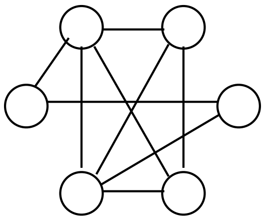
A *clique* in an undirected graph G is a subgraph H such that for every pair of distinct nodes in H there is an edge in G. A *k-clique* is a clique containing k nodes.

The decision problem: *Given an undirected graph G and a natural number k, does G contain a clique of size k?*

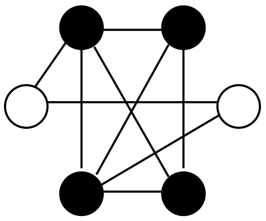
The corresponding language:

$$\mathit{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is a undirected graph having a } k\text{-clique}\}$$

Example: Does this graph have a 4-clique?



Yes:



Theorem. $\mathit{CLIQUE} \in$ NP.

Proof. To verify membership in this language, use a certificate consisting of a list of k distinct nodes that are claimed to form a k-clique. Verification requires checking that every pair of distinct nodes in the given certificate has an edge in G. Let m denote the number of nodes in G. Since $\langle G, k\rangle$ should obviously be rejected if $k > m$, the only interesting case to consider is when $k \leq m$. In this case the number of pairs of distinct nodes in the certificate must be $O(m^2)$. Checking that each of these has an edge in G can obviously be done in polynomial time, so checking all $O(m^2)$ of them can be done in polynomial time.

The Vertex Cover Problem

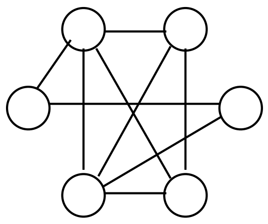
Given an undirected graph G, a *vertex cover* of G is a set S of nodes such that every edge in G is connected to at least one node in S.

The decision problem: *Given an undirected graph G and a natural number k, does G have a vertex cover of size k?*

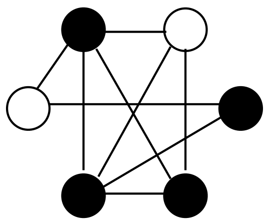
The corresponding language:

$$\mathit{VERTEX\text{-}COVER} = \{\langle G, k\rangle \mid G \text{ is a undirected graph having a } k\text{-node vertex cover}\}$$

Example: Does this graph have a 4-node vertex cover?



Yes:



Theorem. *VERTEX-COVER* $\in$ NP.

Proof. To verify membership in this language, use a certificate consisting of a list of k distinct nodes that are claimed to form a vertex cover. Verification consists simply of checking that every edge does, indeed, include at least one node in the given list, and this can obviously be done in polynomial time.

Factoring Integers

An integer is *composite* if it is not prime – i.e. if it has any integer factors other than itself and 1.

The decision problem: *Given a natural number k, is k composite?*

The corresponding language:

$$COMPOSITES = \{\langle k\rangle \mid k = pq \text{ where } p \text{ and } q \text{ are integers} > 1\}$$

Brute-force approach: Try every integer p from 2 up to[1] $\sqrt{k}$ and see if it divides evenly into k. The number of tests is $O(\sqrt{k})$, which is exponential in the length n of the base-b representation of k for $b \geq 2$. This is not a polytime algorithm.

Theorem. $COMPOSITES \in$ NP.

Proof. Use one of the factors as the certificate. Verification involves simply performing the integer division and seeing whether the remainder is zero, which can be done in polynomial time.

[1]The largest integer $\leq$ the square root of an integer can be found by a polytime algorithm, but a simpler way to get a reasonable upper bound on this square root is to extract the left half of its base-b representation (for $b \geq 2$).

The Subset Sum Problem

The decision problem: *Given a set of numbers S and a number t, does S contain a subset of numbers that add up to t?*

The corresponding language:

$$\mathit{SUBSET\text{-}SUM} =$$

$$\{\langle S, t\rangle \mid S \text{ is a set of numbers containing a subset of numbers that add up to } t\}$$

For example, $\langle\{4, 11, 12, 20, 30\}, 35\rangle \in \mathit{SUBSET\text{-}SUM}$ since $4 + 11 + 20 = 35$.

Brute-force approach: Try every subset and see if the sum for any of these matches the target value. If there are m elements in the set, this means testing $O(2^m)$ subsets.

Theorem. $\mathit{SUBSET\text{-}SUM} \in \mathrm{NP}$.

Proof. Use the subset as the certificate. Verification involves simply performing the addition and comparing the result to the desired target sum.

Satisfiability of Boolean Formulas

A Boolean formula is a legal expression involving

- Boolean constants

  0 (representing FALSE)
  1 (representing TRUE)

- Boolean variables
- AND (written as $\wedge$)
- OR (written as $\vee$)
- NOT (written using overbar)

A Boolean variable can take on one of two possible values:

- 0 (representing FALSE)
- 1 (representing TRUE)

Given any assignment of 0/1 values to each of its variables, a Boolean formula can then be evaluated, yielding either 0 or 1. The value of the formula is determined through the use of these familiar truth tables:

x	$\overline{x}$
0	1
1	0

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Boolean formulas also use parentheses for grouping, but they can be omitted when there's no chance of ambiguity. For example, $\wedge$ and $\vee$ are associative and this allows omission of parentheses in certain expressions without leading to ambiguity.

Example of a Boolean formula: $\phi = (x_1 \wedge x_2) \vee \overline{(x_1 \vee \overline{x_3})} \vee x_3$

The assignment of values $x_1 = 0$, $x_2 = 1$, $x_3 = 0$ makes ϕ evaluate to

$$(0 \wedge 1) \vee \overline{(0 \vee \overline{0})} \vee 0 = 0 \vee \overline{(0 \vee 1)} \vee 0 = 0 \vee \overline{1} \vee 0 = 0 \vee 0 \vee 0 = 0.$$

An assignment of values to the variables of a Boolean formula that cause it to evaluate to 1 is called a *satisfying assignment* for that formula. A formula is *satisfiable* if it has a satisfying assignment.

The above formula ϕ is satisfiable because it is easy to check that $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ is a satisfying assignment for it.

Example of an unsatifiable formula: $\phi = x \wedge \overline{x}$

Satisfiability of Boolean Formulas (Continued)

The satisfiability problem: *Given a Boolean formula ϕ, is ϕ satisfiable?*

The corresponding language:

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

Brute-force (deterministic) decider for SAT: Try every possible assignment of 0/1 values to all the variables.

If there are m variables, there are 2^m possible assignments of values to try. Therefore this is not a polytime decider.

Theorem. $SAT \in$ NP.

Proof. We use a polytime verifier for which the certificate c is an assignment of 0/1 values to all the variables.

```
$V_{SAT}$ =  "On input $\langle\langle\phi\rangle, c\rangle$:
     1.   Evaluate $\phi$ by substituting for all its variables according to the assignment $c$.
     2.   If the resulting value is 1, accept; otherwise, reject."
```

Evaluation of a Boolean formula can be done in time polynomial in the size of the formula. This is so because once the 0/1 values have been substituted in, the expression can be simplified one step at a time just as was done in the example on the previous page. The number of such steps is on the order of the size of the formula (since each step eliminates at least one of the Boolean operators $\wedge$, $\vee$, and negation from the expression), and each step can easily be done in polynomial time.

3SAT

Now we define what turns out to be a useful sublanguage of SAT in which all the formulas have a special form.

First:

- A *literal* is a single Boolean variable or its negation, as in x or $\overline{x}$.
- A *clause* consists of one or more literals joined by $\vee$ operators, as in $x_1 \vee \overline{x_3}$ or $x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4}$.
- A Boolean formula is in *conjunctive normal form (CNF)* if it consists of one or more clauses joined by $\wedge$ operators, as in
$$(x_1 \vee \overline{x_2} \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3}) \wedge x_1.$$
- A Boolean formula is a 3CNF formula if it is in CNF and every clause has exactly 3 literals, as in
$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_3 \vee x_3 \vee x_4) \wedge (\overline{x_2} \vee x_3 \vee x_4).$$

The decision problem: *Given a 3CNF formula ϕ, is ϕ satisfiable?*

The corresponding language:

$$\mathit{3SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3CNF Boolean formula}\}$$

Theorem. *3SAT* $\in$ NP.

Proof. Clearly the verifier V_{SAT} is a polytime verifier for *3SAT* as well.

Useful observation: To be a satisfying assignment for any CNF formula, an assignment must make every clause TRUE; if any single clause is FALSE, the whole formula is FALSE. Furthermore, any clause is TRUE exactly when at least one of its literals is TRUE.

The $64,000 Question

Here's a list of all the languages shown to be in NP in this handout:

HAMPATH

UHAMPATH

CLIQUE

VERTEX-COVER

COMPOSITES

SUBSET-SUM

SAT

3SAT

Are any of the languages on this list decidable in polynomial time?

With the exception of *COMPOSITES*, for which a polytime algorithm was first discovered in 2002, no one knows. A lot of research has been done on these and many other languages in NP to try to either:

- find a polytime algorithm for them; or
- prove that no such polytime algorithm exists.

This is where the theory of *NP-completeness* comes in ...