# The classes FNP and TFNP

C. Wilson[1]

[1] Lane Department of Computer Science and Electrical Engineering
West Virginia University

# Outline

1. Function Problems defined
   - What are Function Problems?
   - FSAT Defined
   - TSP Defined
2. Relationship between Function and Decision Problems
   - $R_L$ Defined
   - Reductions between Function Problems
3. Total Functions Defined
   - Total Functions Defined
   - FACTORING
   - HAPPYNET
   - ANOTHER HAMILTON CYCLE

# Outline

# Outline

# Outline

# What are Function Problems?

## Query

What are Function Problems?

## Definition

Function problems are problems that require an answer more sophisticated than a "yes" or "no" given by a decision problem.

## Example

(i) Satisfying a boolean expression
(ii) Traveling salesman: the actual tour

# What are Function Problems?

**Query**

What are Function Problems?

**Definition**

Function problems are problems that require an answer more sophisticated than a "yes" or "no" given by a decision problem.

**Example**

(i) Satisfying a boolean expression
(ii) Traveling salesman: the actual tour

# What are Function Problems?

**Query**

What are Function Problems?

**Definition**

Function problems are problems that require an answer more sophisticated than a "yes" or "no" given by a decision problem.

**Example**

(i) Satisfying a boolean expression
(ii) Traveling salesman: the actual tour

# What are Function Problems?

**Query**

What are Function Problems?

**Definition**

Function problems are problems that require an answer more sophisticated than a "yes" or "no" given by a decision problem.

**Example**

(i) Satisfying a boolean expression
(ii) Traveling salesman: the actual tour

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems

(ii) Useful in the context of negative complexity results

(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.

(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.

(v) However, for the examples presented here today, this is not the case.

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems

(ii) Useful in the context of negative complexity results

(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.

(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.

(v) However, for the examples presented here today, this is not the case.

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems

(ii) Useful in the context of negative complexity results

(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.

(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.

(v) However, for the examples presented here today, this is not the case.

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems
(ii) Useful in the context of negative complexity results
(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.
(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.
(v) However, for the examples presented here today, this is not the case.

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems
(ii) Useful in the context of negative complexity results
(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.
(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.
(v) However, for the examples presented here today, this is not the case.

# Function Problems Vs. Decision Problems

## More on Function Problems

(i) Decision problems are often considered surrogates for Function problems
(ii) Useful in the context of negative complexity results
(iii) Decisions are often used to show a problem is NP-complete. Unless P = NP, then no polynomial solution exists.
(iv) It is also important to note that a decision problem could be significantly easier to compute than their function counterpart.
(v) However, for the examples presented here today, this is not the case.

# Outline

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If "no", stop. If "yes", continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, "flip" $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\rightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If ”no”, stop. If ”yes”, continue.

(ii) For each $x_i$ perform a truth assignment.

(iii) If successful, move on to statement $x_{i+1}$.

(iv) If unsuccessful, ”flip” $x_i$ and move on.

(v) Worse case: $2n$ calls to SAT.

(vi) If SAT in P $\rightarrow$ FSAT in P.

(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If "no", stop. If "yes", continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, "flip" $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\rightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If ”no”, stop. If ”yes”, continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, ”flip” $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\rightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If ”no”, stop. If ”yes”, continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, ”flip” $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\rightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If "no", stop. If "yes", continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, "flip" $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\longrightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\longrightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If ”no”, stop. If ”yes”, continue.
(ii) For each $x_i$ perform a truth assignment.
(iii) If successful, move on to statement $x_{i+1}$.
(iv) If unsuccessful, ”flip” $x_i$ and move on.
(v) Worse case: $2n$ calls to SAT.
(vi) If SAT in P $\longrightarrow$ FSAT in P.
(vii) Likewise, FSAT in P $\longrightarrow$ SAT in P.

# FSAT Defined

## Definition

Problem Statement: Given an expression $\phi$ with variables $x_1, x_2, \ldots, x_n$, if $\phi$ is satisfiable, return a satisfying truth assignment, otherwise return no.

## FSAT Solution

(i) Test for satisfiability (Call SAT). If ”no”, stop. If ”yes”, continue.

(ii) For each $x_i$ perform a truth assignment.

(iii) If successful, move on to statement $x_{i+1}$.

(iv) If unsuccessful, ”flip” $x_i$ and move on.

(v) Worse case: $2n$ calls to SAT.

(vi) If SAT in P $\rightarrow$ FSAT in P.

(vii) Likewise, FSAT in P $\rightarrow$ SAT in P.

# FSAT Final Thought

## FSAT Final Thought

FSAT uses the self-reducing properties of SAT, like many other NP-complete problems.

# Outline

1. Function Problems defined
   - What are Function Problems?
   - FSAT Defined
   - TSP Defined
2. Relationship between Function and Decision Problems
   - $R_L$ Defined
   - Reductions between Function Problems
3. Total Functions Defined
   - Total Functions Defined
   - FACTORING
   - HAPPYNET
   - ANOTHER HAMILTON CYCLE

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).
(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.
(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.
(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.
(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).
(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.
(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.
(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.
(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).
(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.
(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.
(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.
(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).

(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.

(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.

(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.

(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).
(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.
(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.
(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.
(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# TSP Definition

## Definition

Problem Statement: Given a graph $G$ with $n$ nodes, and a cost $C$, find out if there is a tour of $G$ that costs exactly $C$.

## TSP Solution

(i) First, find optimal cost C, by performing a binary search and using TSP(D).
(ii) Next, select any path and set the cost to $C + 1$. Perform TSP(D) with $C$ or less.
(iii) If TSP(D) returns yes, It is not part of optimal tour. Freeze its cost at $C + 1$.
(iv) If TSP(D) returns no, we know the path we are considering is crucial to optimum path.
(v) Call TSP(D) $n^2$ times to process everything; eliminate all but the critical path.

# Outline

# $R_L$ Defined

**What is $R_L$?**

The relationship between function and decision problems can be formalized.
This relation is known as $R_L$.

# $R_L$ Defined

**What is $R_L$?**

The relationship between function and decision problems can be formalized. This relation is known as $R_L$.

# $R_L$ Defined

## $R_L$ Formally Defined

(a) Suppose that $L$ is a language in $NP$.

(b) There is a polynomial-time decidable, polynomial balanced relation $R_L$ such that for all strings $x$ :

(c) There $\exists y \forall x$ with $R_L(x, y)$ if and only if $x \in L$.

# $R_L$ Defined

## $R_L$ Formally Defined

(a) Suppose that $L$ is a language in $NP$.

(b) There is a polynomial-time decidable, polynomial balanced relation $R_L$ such that for all strings $x$ :

(c) There $\exists y \forall x$ with $R_L(x, y)$ if and only if $x \in L$.

# $R_L$ Defined

## $R_L$ Formally Defined

(a) Suppose that $L$ is a language in $NP$.
(b) There is a polynomial-time decidable, polynomial balanced relation $R_L$ such that for all strings $x$ :
(c) There $\exists y \forall x$ with $R_L(x, y)$ if and only if $x \in L$.

# $R_L$ Defined

## $R_L$ Formally Defined

(a) Suppose that $L$ is a language in $NP$.

(b) There is a polynomial-time decidable, polynomial balanced relation $R_L$ such that for all strings $x$ :

(c) There $\exists y \forall x$ with $R_L(x, y)$ if and only if $x \in L$.

# *FL* Defined

**Definition (FL)**

The Function problem associated with $L$, denoted $FL$, is the following computational problem:
Given $x$, find a string $y$ such that $R_L(x, y)$ if such a string exists; if no such string exists, return no.

**Definition (FNP)**

The class of all function problems associates as above with languages in $NP$ is called $FNP$.

**Definition (FP)**

The subclass of $FNP$ that can be solved in polynomial time.

# $FL$ Defined

**Definition (FL)**

The Function problem associated with $L$, denoted $FL$, is the following computational problem:
Given $x$, find a string $y$ such that $R_L(x, y)$ if such a string exists; if no such string exists, return no.

**Definition (FNP)**

The class of all function problems associates as above with languages in $NP$ is called $FNP$.

**Definition (FP)**

The subclass of $FNP$ that can be solved in polynomial time.

# *FL* Defined

**Definition (FL)**

The Function problem associated with $L$, denoted $FL$, is the following computational problem:
Given $x$, find a string $y$ such that $R_L(x, y)$ if such a string exists; if no such string exists, return no.

**Definition (FNP)**

The class of all function problems associates as above with languages in $NP$ is called $FNP$.

**Definition (FP)**

The subclass of $FNP$ that can be solved in polynomial time.

# *FL* Defined

**Definition (FL)**

The Function problem associated with $L$, denoted $FL$, is the following computational problem:
Given $x$, find a string $y$ such that $R_L(x, y)$ if such a string exists; if no such string exists, return no.

**Definition (FNP)**

The class of all function problems associates as above with languages in $NP$ is called $FNP$.

**Definition (FP)**

The subclass of $FNP$ that can be solved in polynomial time.

# $FL$ Defined

**Definition (FL)**

The Function problem associated with $L$, denoted $FL$, is the following computational problem:
Given $x$, find a string $y$ such that $R_L(x, y)$ if such a string exists; if no such string exists, return no.

**Definition (FNP)**

The class of all function problems associates as above with languages in $NP$ is called $FNP$.

**Definition (FP)**

The subclass of $FNP$ that can be solved in polynomial time.

# *FNP* and *FP* Examples

**Example (FNP Example)**

FSAT is in FNP.

**Example (FP Example)**

*FHORNSAT* is in *FP*.
Finding a match in a bipartite graph is in *FP*.

**Note**

We do not say that TSP is in *FNP* because it probably isn't. The reason is, in the case of TSP, the optimal solution is not an adequate certificate, as we do not know how to verify in polynomial time that it is optimal.

# *FNP* and *FP* Examples

**Example (FNP Example)**

FSAT is in FNP.

**Example (FP Example)**

*FHORNSAT* is in *FP*.
Finding a match in a bipartite graph is in *FP*.

**Note**

We do not say that TSP is in *FNP* because it probably isn't. The reason is, in the case of TSP, the optimal solution is not an adequate certificate, as we do not know how to verify in polynomial time that it is optimal.

# *FNP* and *FP* Examples

**Example (FNP Example)**

FSAT is in FNP.

**Example (FP Example)**

*FHORNSAT* is in *FP*.
Finding a match in a bipartite graph is in *FP*.

**Note**

We do not say that TSP is in *FNP* because it probably isn't. The reason is, in the case of TSP, the optimal solution is not an adequate certificate, as we do not know how to verify in polynomial time that it is optimal.

# Outline

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.

Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.

We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.

$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.

Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.

We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.

$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

*$FP = FNP$ if and only if $P = NP$.*

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.

Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.

We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.

$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ *if and only if* $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.

Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.

We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.

$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.
Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.
We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.
$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.
Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.
We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.
$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.
Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.
We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.
$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.
Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.
We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.
*FP* and *FNP* are closed under reductions. *FSAT* is *FNP* complete.

## Theorem

*$FP = FNP$ if and only if $P = NP$.*

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.

Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.

We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.

$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

$FP = FNP$ if and only if $P = NP$.

# Reductions between Function Problems

## Reductions

We say that a function problem $A$ reduces to function problem $B$ if the following logic holds: There are string functions $R$ and $S$, both computable in $log(n)$ space, such that for any strings $x$ and $z$ the following holds: If $x$ is an instance of $A$ then $R(x)$ is an instance of $B$. Futhermore, if $z$ is a correct output of $R(x)$, then $S(z)$ is a correct output of $x$.
Notice: $R$ produces an instance $R(x)$ of the function problem $B$ such that we can construct an output $S(z)$ for $x$ from any correct output $z$ of $R(x)$.
We say that a function problem $A$ is complete for a class $FC$ of function problems if it $\in FC$ and all problems in that class reduce to $A$.
$FP$ and $FNP$ are closed under reductions. $FSAT$ is $FNP$ complete.

## Theorem

*$FP = FNP$ if and only if $P = NP$.*

# Outline

# Total Functions

## Total Functions

There are some Function Problems that have no decision counterpart or no meaningful language equivalent.

These problems in *FNP* are somehow guaranteed to never return no. A function is said to be *total* if for every problem size, a reasonable decomposition exists.

# Total Functions

## Total Functions

There are some Function Problems that have no decision counterpart or no meaningful language equivalent.
These problems in *FNP* are somehow guaranteed to never return no. A function is said to be *total* if for every problem size, a reasonable decomposition exists.

# Total Functions – Examples

## Examples of Total Functions

The following are famous examples of total functions within *FNP* space.

(i) FACTORING
(ii) HAPPYNET
(iii) ANOTHER HAMILTON CYCLE

# Total Functions – Examples

## Examples of Total Functions

The following are famous examples of total functions within *FNP* space.

(i) FACTORING

(ii) HAPPYNET

(iii) ANOTHER HAMILTON CYCLE

# Total Functions – Examples

## Examples of Total Functions

The following are famous examples of total functions within *FNP* space.

(i) FACTORING
(ii) HAPPYNET
(iii) ANOTHER HAMILTON CYCLE

# Total Functions – Examples

## Examples of Total Functions

The following are famous examples of total functions within *FNP* space.

(i) FACTORING
(ii) HAPPYNET
(iii) ANOTHER HAMILTON CYCLE

# Outline

# FACTORING example

## Example

Problem Statement: Given an integer $N$, find its prime decomposition $N = p_1^{k_1}, p_2^{k_2}, \ldots, p_m^{k_m}$ together with its primality certificates $p_1, p_2, \ldots p_m$.

Notice the requirement that the output includes the certificates of the prime divisors; without it the problem would not be in *FNP*.

Despite serious efforts, no polynomial algorithm for FACTORING is known.

It is plausible (but not universally believed) that there is no polynomial algorithm for FACTORING.

# FACTORING example

## Example

Problem Statement: Given an integer $N$, find its prime decomposition $N = p_1^{k_1}, p_2^{k_2}, \ldots, p_m^{k_m}$ together with its primality certificates $p_1, p_2, \ldots p_m$.
Notice the requirement that the output includes the certificates of the prime divisors; without it the problem would not be in *FNP*.
Despite serious efforts, no polynomial algorithm for FACTORING is known.
It is plausible (but not universally believed) that there is no polynomial algorithm for FACTORING.

# FACTORING example

## Example

Problem Statement: Given an integer $N$, find its prime decomposition $N = p_1^{k_1}, p_2^{k_2}, \ldots, p_m^{k_m}$ together with its primality certificates $p_1, p_2, \ldots p_m$.
Notice the requirement that the output includes the certificates of the prime divisors; without it the problem would not be in *FNP*.
Despite serious efforts, no polynomial algorithm for FACTORING is known.
It is plausible (but not universally believed) that there is no polynomial algorithm for FACTORING.

# FACTORING example

## Example

Problem Statement: Given an integer $N$, find its prime decomposition $N = p_1^{k_1}, p_2^{k_2}, \ldots, p_m^{k_m}$ together with its primality certificates $p_1, p_2, \ldots p_m$.
Notice the requirement that the output includes the certificates of the prime divisors; without it the problem would not be in *FNP*.
Despite serious efforts, no polynomial algorithm for FACTORING is known.
It is plausible (but not universally believed) that there is no polynomial algorithm for FACTORING.

# Outline

# HAPPYNET example

**Example**

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to { -1, +1}.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i,j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.

# HAPPYNET example

## Example

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to { -1, +1}.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i,j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.

# HAPPYNET example

## Example

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to $\{$ -1, +1$\}$.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i,j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.

# HAPPYNET example

## Example

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to $\{$ -1, +1$\}$.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i, j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.

# HAPPYNET example

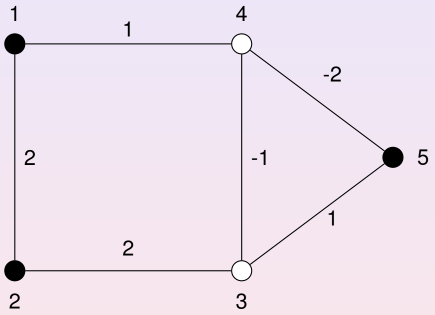
## Example

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to { -1, +1}.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i,j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.

# HAPPYNET example

## Example

Problem Statement: We are given an undirected graph $G(V, E)$ with integer (possibly negative) weights $w$ on its edges.
The nodes are "people", and the edge weight an indication of how much (or how little) these two people like each other.
Assume it to be symmetric.
We define state $S$ as a mapping from $V$ to { -1, +1}.
We say node $i$ is *happy* in state $S$ if the following holds:
$S(i) * \sum S(j)w[i, j] \geq 0$. Where $i$ and $j \in E$.
We want to find a state in which all nodes are happy.


1
4
1
-2
2
-1
5
1
2
2
3

# HAPPYNET continued

## HAPPYNET continued

The ”happy state” conveys that a node prefers to have the same value of an adjacent node to which its connected through a positive edge, and the opposite value from a node adjacent via a negative edge.

At first, this seems like a typical hard combinatorial problem.

There is no known polynomial-time algorithm for finding a happy state.

However, all instances of HAPPYNET are guaranteed to have a solution. (see book for proof).

# HAPPYNET continued

## HAPPYNET continued

The "happy state" conveys that a node prefers to have the same value of an adjacent node to which its connected through a positive edge, and the opposite value from a node adjacent via a negative edge.
At first, this seems like a typical hard combinatorial problem.
There is no known polynomial-time algorithm for finding a happy state.
However, all instances of HAPPYNET are guaranteed to have a solution. (see book for proof).

# HAPPYNET continued

## HAPPYNET continued

The "happy state" conveys that a node prefers to have the same value of an adjacent node to which its connected through a positive edge, and the opposite value from a node adjacent via a negative edge.
At first, this seems like a typical hard combinatorial problem.
There is no known polynomial-time algorithm for finding a happy state.
However, all instances of HAPPYNET are guaranteed to have a solution. (see book for proof).

# HAPPYNET continued

## HAPPYNET continued

The "happy state" conveys that a node prefers to have the same value of an adjacent node to which its connected through a positive edge, and the opposite value from a node adjacent via a negative edge.
At first, this seems like a typical hard combinatorial problem.
There is no known polynomial-time algorithm for finding a happy state.
However, all instances of HAPPYNET are guaranteed to have a solution. (see book for proof).

# HAPPYNET Solution

## HAPPYNET Solution

The solution to HAPPYNET is iterative.
We define $S'(i) = -S(i)$. We say that $i$ was "flipped".
We start with any state $S$, and repeat:
Whe there is an unhappy node, flip it.
HAPPYNET $\in$ *TFNP*.

# HAPPYNET Solution

## HAPPYNET Solution

The solution to HAPPYNET is iterative.
We define $S'(i) = -S(i)$. We say that $i$ was "flipped".
We start with any state $S$, and repeat:
Whe there is an unhappy node, flip it.
HAPPYNET $\in$ *TFNP*.

# HAPPYNET Solution

## HAPPYNET Solution

The solution to HAPPYNET is iterative.
We define $S'(i) = -S(i)$. We say that $i$ was "flipped".
We start with any state $S$, and repeat:
Whe there is an unhappy node, flip it.
HAPPYNET $\in$ *TFNP*.

# HAPPYNET Solution

## HAPPYNET Solution

The solution to HAPPYNET is iterative.
We define $S'(i) = -S(i)$. We say that $i$ was "flipped".
We start with any state $S$, and repeat:
Whe there is an unhappy node, flip it.
HAPPYNET $\in$ *TFNP*.

# Outline

# ANOTHER HAMILTON CYCLE

**Example (Problem Statement)**

We know that it is NP-complete, given a graph, to find a Hamilton cycle. But what if a Hamilton cycle is given, and we are asked to find another Hamilton cycle? The existing cycle should facilitate our search for the new one.

This problem, ANOTHER HAMILTON CYCLE, is $FNP-complete$.

# ANOTHER HAMILTON CYCLE

**Example (Problem Statement)**

We know that it is NP-complete, given a graph, to find a Hamilton cycle. But what if a Hamilton cycle is given, and we are asked to find another Hamilton cycle? The existing cycle should facilitate our search for the new one.
This problem, ANOTHER HAMILTON CYCLE, is $FNP-complete$.

# ANOTHER HAMILTON CYCLE

**Example (Problem Statement)**

We know that it is NP-complete, given a graph, to find a Hamilton cycle. But what if a Hamilton cycle is given, and we are asked to find another Hamilton cycle? The existing cycle should facilitate our search for the new one.
This problem, ANOTHER HAMILTON CYCLE, is $FNP - complete$.

# ANOTHER HAMILTON CYCLE

**Example (Problem Statement)**

We know that it is NP-complete, given a graph, to find a Hamilton cycle. But what if a Hamilton cycle is given, and we are asked to find another Hamilton cycle? The existing cycle should facilitate our search for the new one.
This problem, ANOTHER HAMILTON CYCLE, is $FNP - complete$.

# ANOTHER HAMILTON CYCLE

**Assertion**

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

**Proof.**

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].

ii Delete the edge [1,2] to obtain a Hamilton path.

iii We shall only consider paths starting with node 1 and that do not use edge [1,2].

iv We call any such Hamilton path a *candidate*.

v We call any two candidate paths *neighbors* if they have $n-2$ edges in common (all but one).

vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

□

# ANOTHER HAMILTON CYCLE

**Assertion**

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

**Proof.**

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].

ii Delete the edge [1,2] to obtain a Hamilton path.

iii We shall only consider paths starting with node 1 and that do not use edge [1,2].

iv We call any such Hamilton path a *candidate*.

v We call any two candidate paths *neighbors* if they have $n-2$ edges in common (all but one).

vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

# ANOTHER HAMILTON CYCLE

## Assertion

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

## Proof.

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].

ii Delete the edge [1,2] to obtain a Hamilton path.

iii We shall only consider paths starting with node 1 and that do not use edge [1,2].

iv We call any such Hamilton path a *candidate*.

v We call any two candidate paths *neighbors* if they have $n-2$ edges in common (all but one).

vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

# ANOTHER HAMILTON CYCLE

## Assertion

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

## Proof.

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].
ii Delete the edge [1,2] to obtain a Hamilton path.
iii We shall only consider paths starting with node 1 and that do not use edge [1,2].
iv We call any such Hamilton path a *candidate*.
v We call any two candidate paths *neighbors* if they have $n-2$ edges in common (all but one).
vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

# ANOTHER HAMILTON CYCLE

## Assertion

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

## Proof.

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3...$n$,1].

ii Delete the edge [1,2] to obtain a Hamilton path.

iii We shall only consider paths starting with node 1 and that do not use edge [1,2].

iv We call any such Hamilton path a *candidate*.

v We call any two candidate paths *neighbors* if they have $n-2$ edges in common (all but one).

vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

# ANOTHER HAMILTON CYCLE

**Assertion**

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

**Proof.**

i. Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].
ii. Delete the edge [1,2] to obtain a Hamilton path.
iii. We shall only consider paths starting with node 1 and that do not use edge [1,2].
iv. We call any such Hamilton path a *candidate*.
v. We call any two candidate paths *neighbors* if they have $n - 2$ edges in common (all but one).
vi. Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

# ANOTHER HAMILTON CYCLE

**Assertion**

Consider the same problem in a cubic graph, one with all degrees equal to three. It turns out that if a cubic graphic has a Hamilton cycle, then it must have a second one as well.

**Proof.**

i Assume we are given a Hamilton Cycle in a cubic graph, e.g. [1,2,3. . . $n$,1].

ii Delete the edge [1,2] to obtain a Hamilton path.

iii We shall only consider paths starting with node 1 and that do not use edge [1,2].

iv We call any such Hamilton path a *candidate*.

v We call any two candidate paths *neighbors* if they have $n - 2$ edges in common (all but one).

vi Each candidate has two neighbors, unless its other endpoint lies on the deleted path [1,2].

□

# ANOTHER HAMILTON CYCLE

## Proof Conclusion

It is now obvious: Since all candidate paths have two neighbors except for those that have endpoints 1, and 2, which have only one neighbor, then there must be an even number of paths WITH endpoints 1 and 2. But any Hamilton path with the addition of edge [1,2] will yield a Hamilton cycle. We conclude there is an even number of Hamilton cycles using edge [1,2], and since we know of one, another must exist.

# ANOTHER HAMILTON CYCLE

## Proof Conclusion

It is now obvious: Since all candidate paths have two neighbors except for those that have endpoints 1, and 2, which have only one neighbor, then there must be an even number of paths WITH endpoints 1 and 2. But any Hamilton path with the addition of edge [1,2] will yield a Hamilton cycle. We conclude there is an even number of Hamilton cycles using edge [1,2], and since we know of one, another must exist.

# ANOTHER HAMILTON CYCLE

## Proof Conclusion

It is now obvious: Since all candidate paths have two neighbors except for those that have endpoints 1, and 2, which have only one neighbor, then there must be an even number of paths WITH endpoints 1 and 2. But any Hamilton path with the addition of edge [1,2] will yield a Hamilton cycle. We conclude there is an even number of Hamilton cycles using edge [1,2], and since we know of one, another must exist.

# The FC Classes