Algorithms and Data Structures

NP-Completeness
(non-examinable)

Running time hierarchy

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Running time hierarchy

Polynomial time

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Efficient algorithms

Efficient algorithms

An algorithm is typically called *efficient* if it runs in polynomial time.

Efficient algorithms

An algorithm is typically called *efficient* if it runs in polynomial time.

For most of the problems we encountered in this course, we were able to design polynomial time algorithms.

Efficient algorithms

An algorithm is typically called *efficient* if it runs in polynomial time.

For most of the problems we encountered in this course, we were able to design polynomial time algorithms.

What was the exception?

Efficient algorithms

Efficient algorithms

Is it possible to design a polynomial-time algorithm for *every* problem?

Efficient algorithms

Is it possible to design a polynomial-time algorithm for *every* problem?

Are there problems for which polynomial-time algorithms do not exist?

Intractable problems

Intractable problems

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Intractable problems

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

Intractable problems

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

For most problems, we cannot have the ideal definition.

Intractable problems

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

For most problems, we cannot have the ideal definition.

There might be some very clever algorithm hiding out there.

Intractable problems

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

For most problems, we cannot have the ideal definition.

There might be some very clever algorithm hiding out there.

Some problems were believed to not be solvable efficient for many years, but then they were proven to be tractable.

Evidence of intractability

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

Evidence of intractability

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

How can we convince ourselves that it is not possible to design efficient algorithms for Problem A?

Evidence of intractability

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

How can we convince ourselves that it is not possible to design efficient algorithms for Problem A?

We tried for a long time and we didn’t manage to come up with one.

Evidence of intractability

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

How can we convince ourselves that it is not possible to design efficient algorithms for Problem A?

We tried for a long time and we didn’t manage to come up with one.

A lot of people that are more clever and experienced than us tried for a long time and did not manage.

Evidence of intractability

Definition: A problem is called intractable if we do now know of any efficient algorithm that solves it.

Ideal definition: A problem is called intractable if it is not possible to design any efficient algorithm that solves it.

How can we convince ourselves that it is not possible to design efficient algorithms for Problem A?

We tried for a long time and we didn't manage to come up with one.

A lot of people that are more clever and experienced than us tried for a long time and did not manage.

If there was an efficient algorithm for Problem A, we could use it to solve many other problems for which we don't have efficient algorithms.

Polynomial Time Reduction

We are given a problem A that we want to solve.

Polynomial Time Reduction

We are given a problem A that we want to solve.

We can *reduce* solving problem A to solving some other problem B.

Polynomial Time Reduction

We are given a problem A that we want to solve.

We can *reduce* solving problem A to solving some other problem B.

Assume that we had an algorithm ALG^{B} for solving problem B.

Polynomial Time Reduction

We are given a problem A that we want to solve.

We can *reduce* solving problem A to solving some other problem B.

Assume that we had an algorithm ALG^B for solving problem B.

We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.

Polynomial Time Reduction

We are given a problem A that we want to solve.

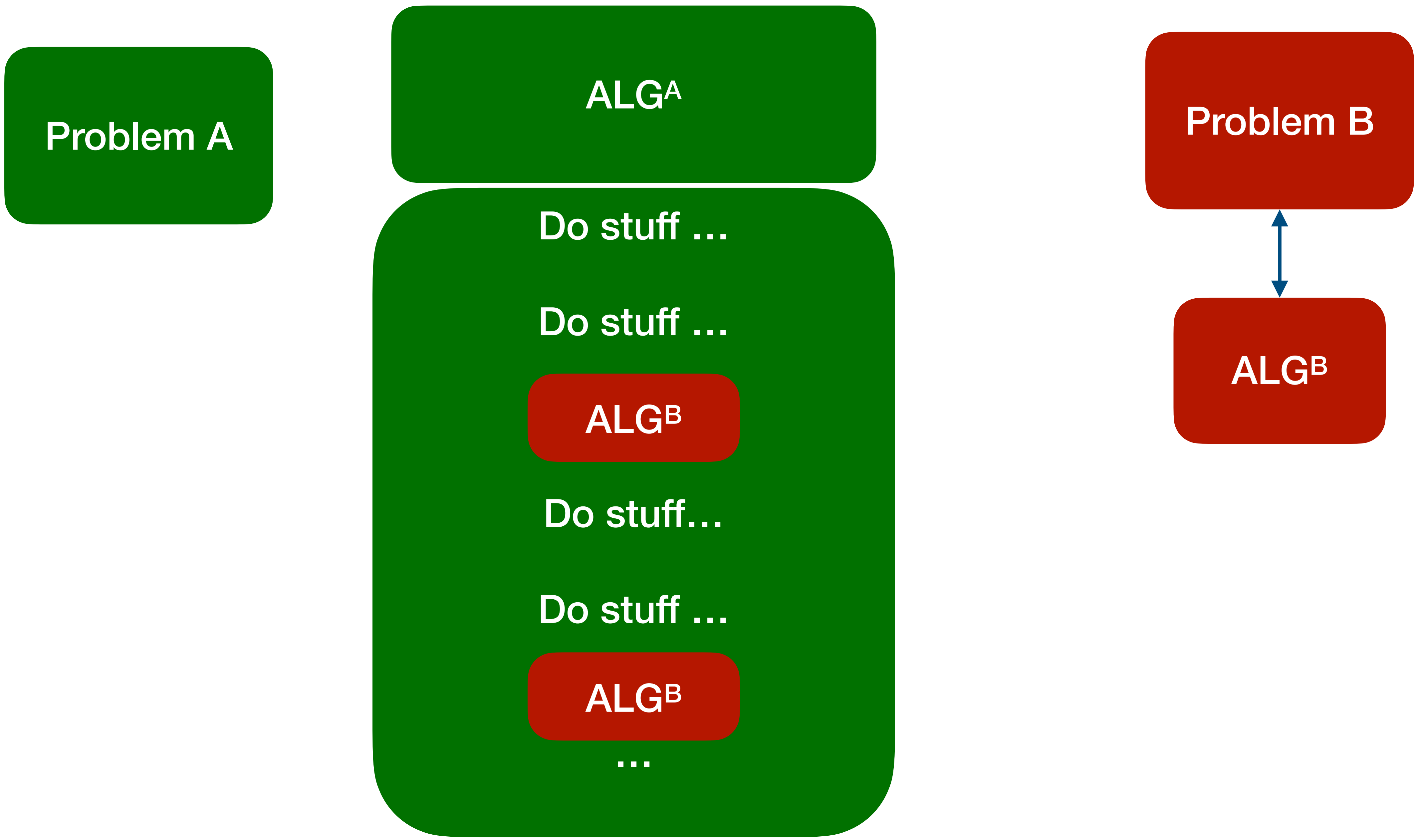
We can *reduce* solving problem A to solving some other problem B.

Assume that we had an algorithm ALG^B for solving problem B.

We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.

If ALG^A is a polynomial time algorithm, then this is a *polynomial time reduction*.

Pictorially



Notation

When problem A reduces to problem B in polynomial time, we write

$A \leq^p B$

We often say “there is a polynomial time reduction *from* A *to* B”.

How to work with reductions

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

Evidence of Intractability

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

Evidence of Intractability

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq_p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

Idea: If we want to provide strong evidence that a problem B cannot be solved by an efficient algorithm, we need to reduce another problem A to it, for which there is strong evidence that it cannot be solved by an efficient algorithm.

Computational classes

Computational classes

Every problem for which there is a known polynomial time algorithm is in the computational class P.

Computational classes

Every problem for which there is a known polynomial time algorithm is in the computational class P.

Searching, sorting, interval scheduling, graph traversal, …

Computational classes

Every problem for which there is a known polynomial time algorithm is in the computational class P.

Searching, sorting, interval scheduling, graph traversal, …

The class P contains computational problems *that can be solved in polynomial time*.

Computational classes

Every problem for which there is a known polynomial time algorithm is in the computational class P.

Searching, sorting, interval scheduling, graph traversal, …

The class P contains computational problems *that can be solved in polynomial time*.

We also say that they can be solved *efficiently*.

The landscape of complexity

contains all problems that
can be solved in polynomial time.

The class NP

The class NP

Stands for "*non deterministic polynomial time*".

The class NP

Stands for “*non deterministic polynomial time*”.

Problems that can be solved in polynomial time by a non-deterministic Turing machine.

The class NP

Stands for "*non deterministic polynomial time*".

Problems that can be solved in polynomial time by a non-deterministic Turing machine.

More intuitive definition:

The class NP

Stands for "*non deterministic polynomial time*".

Problems that can be solved in polynomial time by a non-deterministic Turing machine.

More intuitive definition:

Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.

The class NP

Stands for "*non deterministic polynomial time*".

Problems that can be solved in polynomial time by a non-deterministic Turing machine.

More intuitive definition:

Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.

Efficiently verifiable.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

(“An AND of ORs”).

Each clause has three literals.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

Computational problem 3SAT : Decide if the input formula ϕ has a satisfying assignment.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

(“An AND of ORs”).

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

Computational problem 3SAT : Decide if the input formula ϕ has a satisfying assignment.

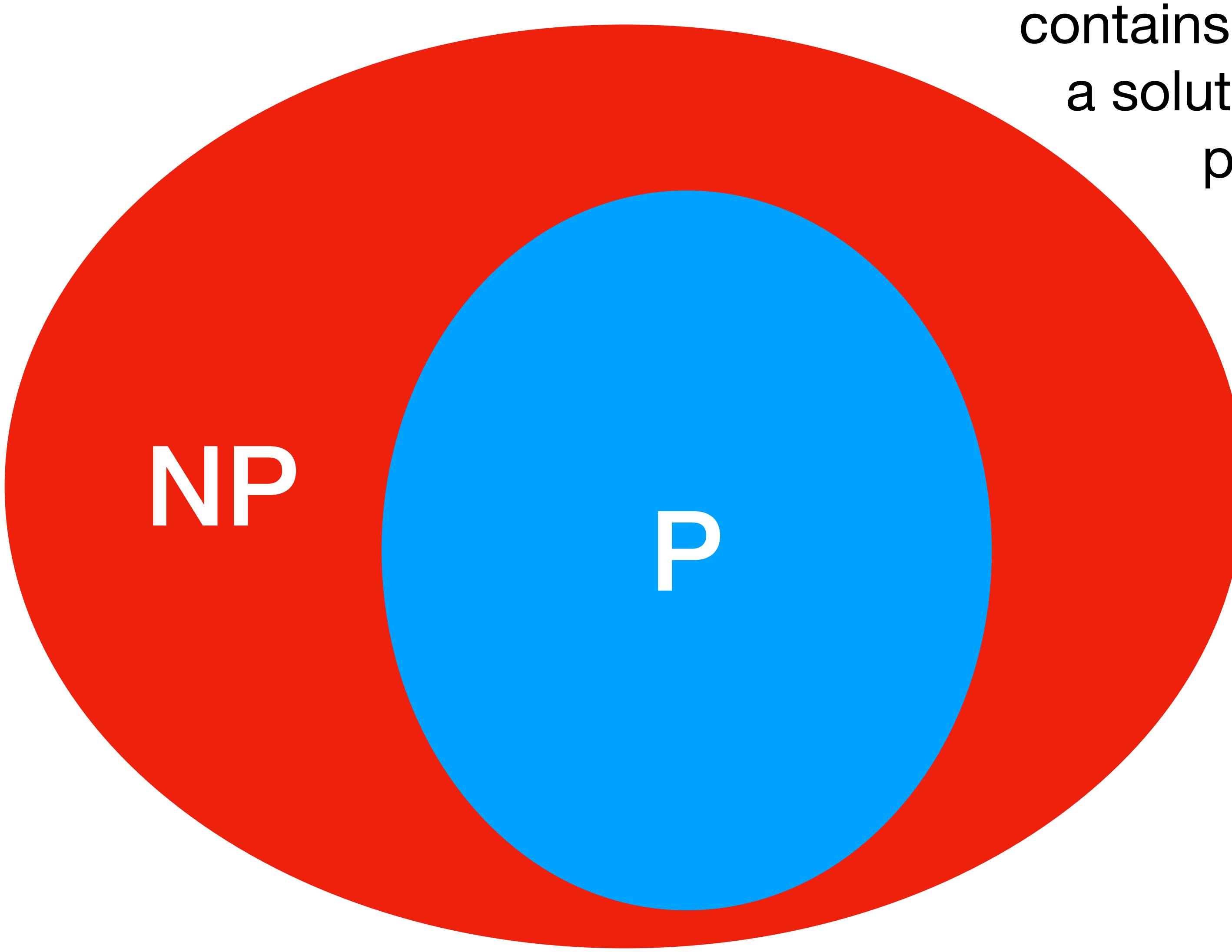
3 SAT is in NP (why?)

The landscape of complexity

contains all problems that
can be solved in polynomial time.

The landscape of complexity

contains all problems for which
a solution can be verified in
polynomial time.



contains all problems that
can be solved in polynomial time.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

How to work with reductions

Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

NP-hardness

A problem B is NP-hard if for every problem A in NP, it holds that A $\leq^p$ B.

If every problem in NP is "polynomial time reducible to B".

This captures the fact that B is *at least as hard as the hardest problems* in NP.

NP-hardness

A problem B is NP-hard if for every problem A in NP, it holds that A $\leq^p$ B.

To prove NP-hardness, it seems that we have to construct a reduction from every problem A in NP.

This is not very useful!

NP-completeness

A problem B is NP-complete if

NP-completeness

A problem B is NP-complete if

It is in NP.

NP-completeness

A problem B is NP-complete if

It is in NP.

i.e., it has a polynomial-time verifiable solution.

NP-completeness

A problem B is NP-complete if

It is in NP.

i.e., it has a polynomial-time verifiable solution.

It is NP-hard.

NP-completeness

A problem B is NP-complete if

It is in NP.

i.e., it has a polynomial-time verifiable solution.

It is NP-hard.

i.e., every problem in NP can be efficiently reduced to it.

NP-completeness

NP-completeness

Assume problem P is NP-complete.

NP-completeness

Assume problem P is NP-complete.

Then every problem in NP is efficiently reducible to P. (why?)

NP-completeness

Assume problem P is NP-complete.

Then every problem in NP is efficiently reducible to P. (why?)

To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.

NP-completeness

Assume problem P is NP-complete.

Then every problem in NP is efficiently reducible to P. (why?)

To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.

Actually, it suffices to construct a reduction from P to B.

NP-completeness

Assume problem P is NP-complete.

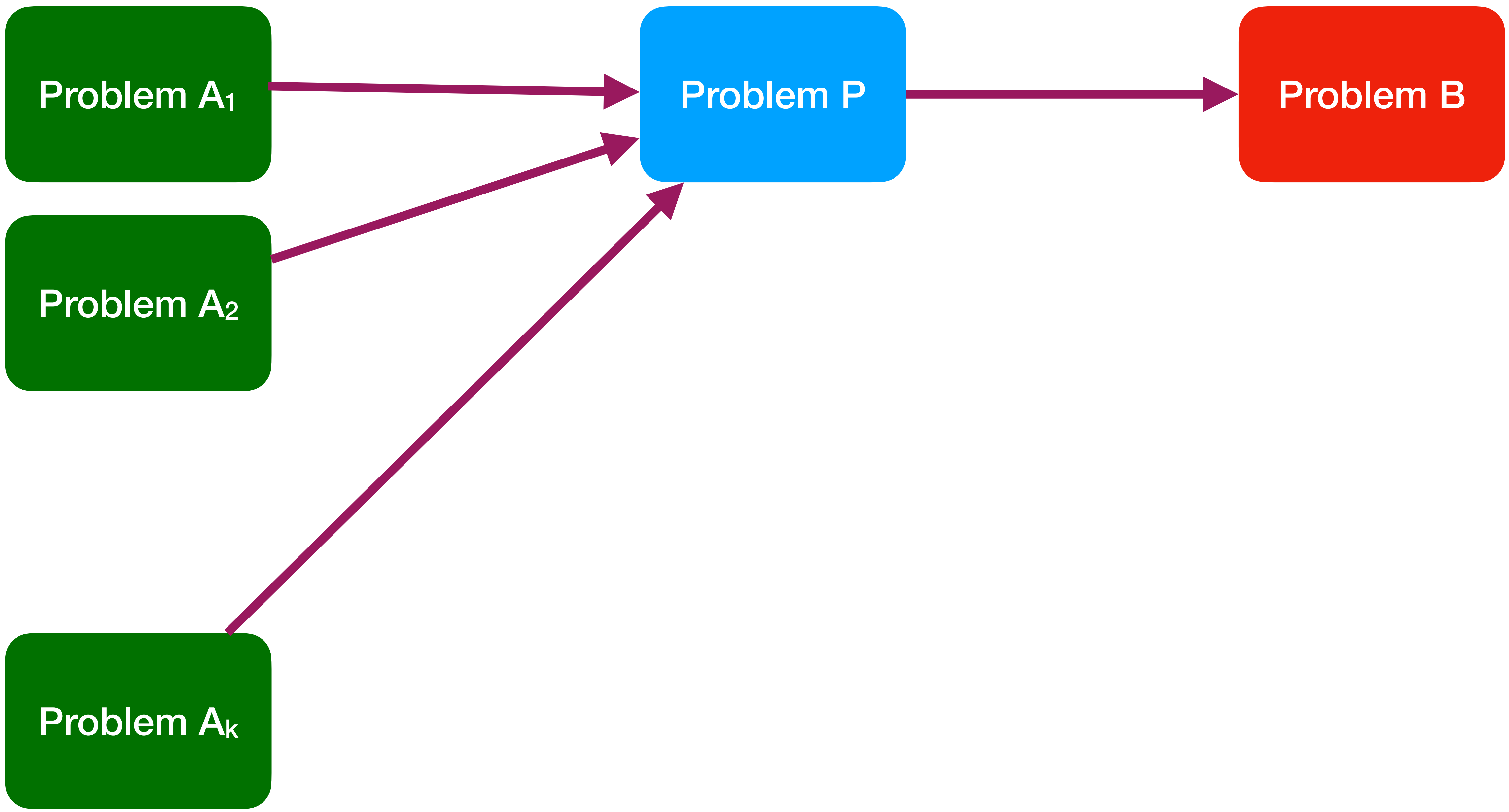
Then every problem in NP is efficiently reducible to P. (why?)

To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.

Actually, it suffices to construct a reduction from P to B.

A reduction from any other problem A to B goes “via” P.

NP-hardness via P



NP-completeness

NP-completeness

Assume problem P is NP-complete.

NP-completeness

Assume problem P is NP-complete.

This all works if we have an NP-complete problem to start with.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

3 SAT

A CNF formula with m clauses and k literals.

$\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

("An AND of ORs").

Each clause has three literals.

Truth assignment: A value in {0,1} for each variable x_i.

Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

Computational problem 3SAT : Decide if the input formula ϕ has a satisfying assignment.

3 SAT is NP-complete

3 SAT is NP-complete

3 SAT is in NP.

3 SAT is NP-complete

3 SAT is in NP.

3 SAT is NP-hard.

3 SAT is NP-complete

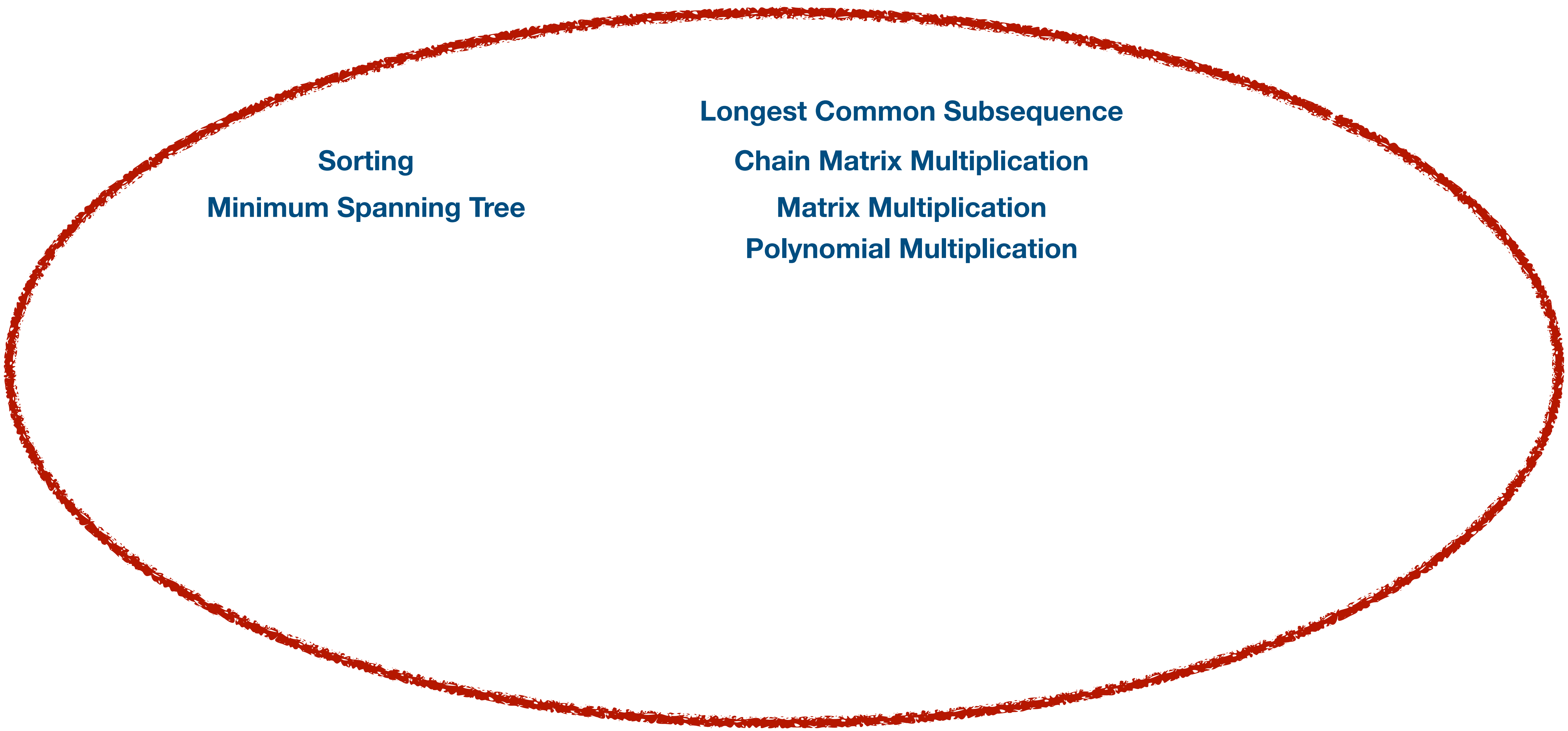
3 SAT is in NP.

3 SAT is NP-hard.

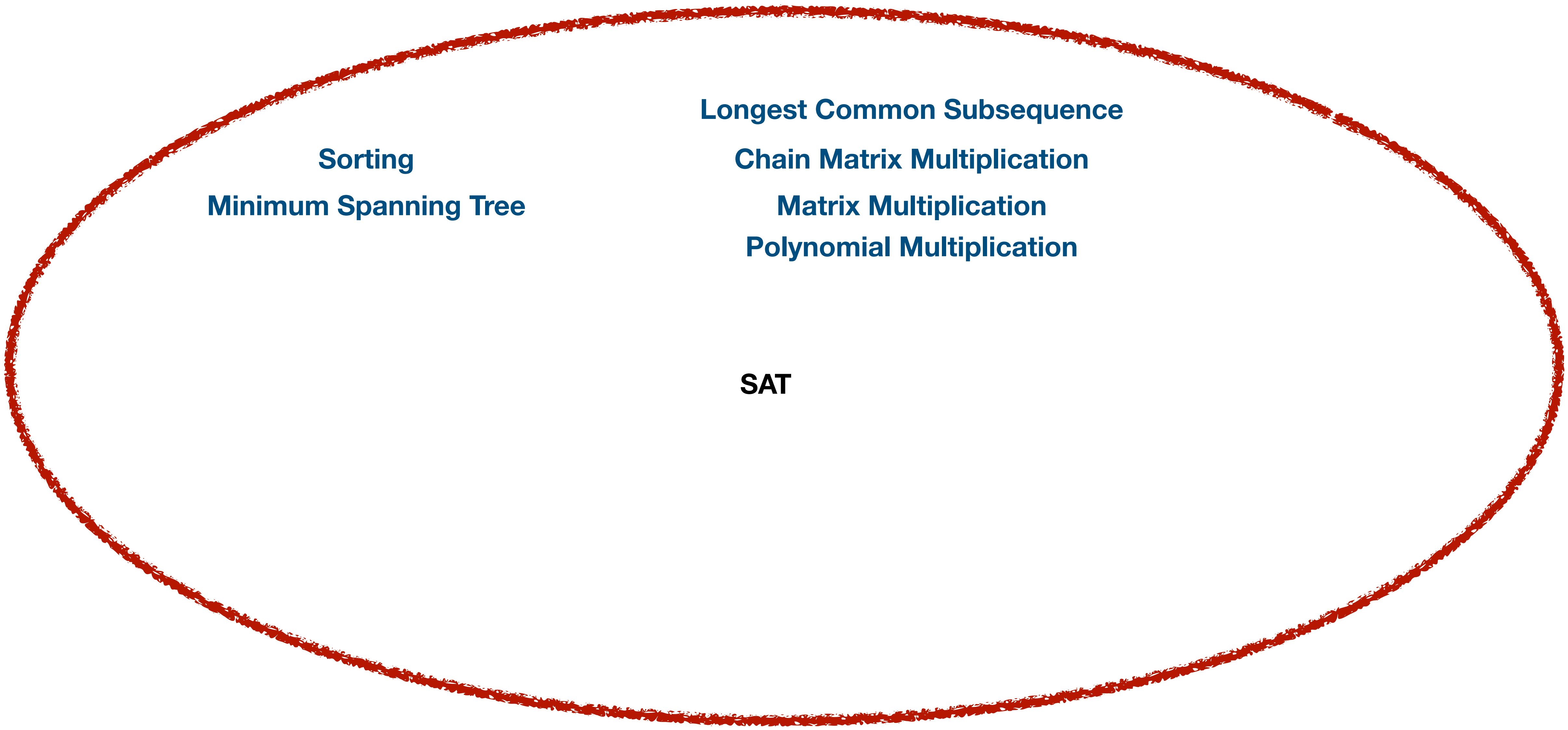
Remark:

The first problem shown to be NP-complete was the SAT problem (more general than 3 SAT, the Cook-Levin Theorem), and this reduces to 3SAT.

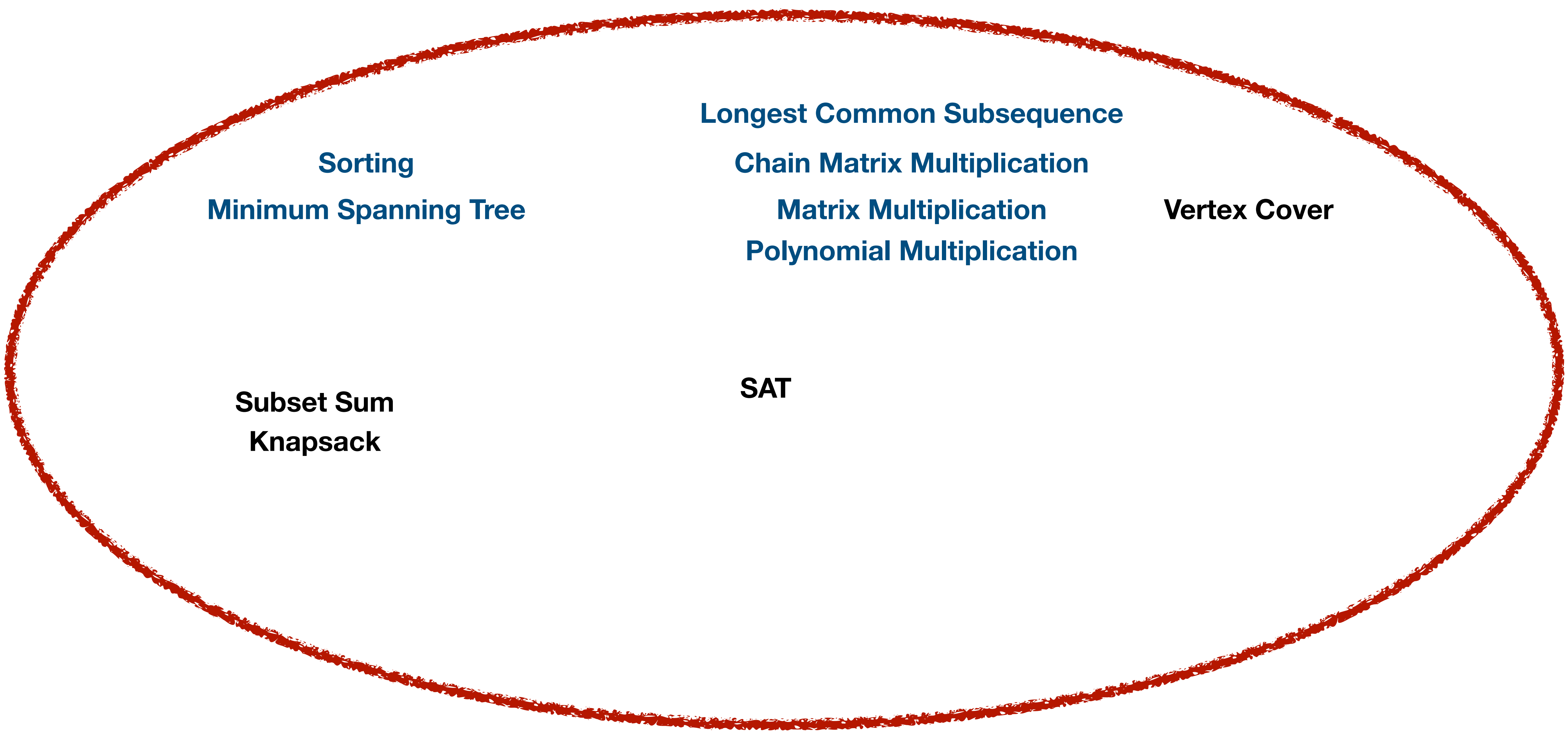
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Polynomial Multiplication

The class NP

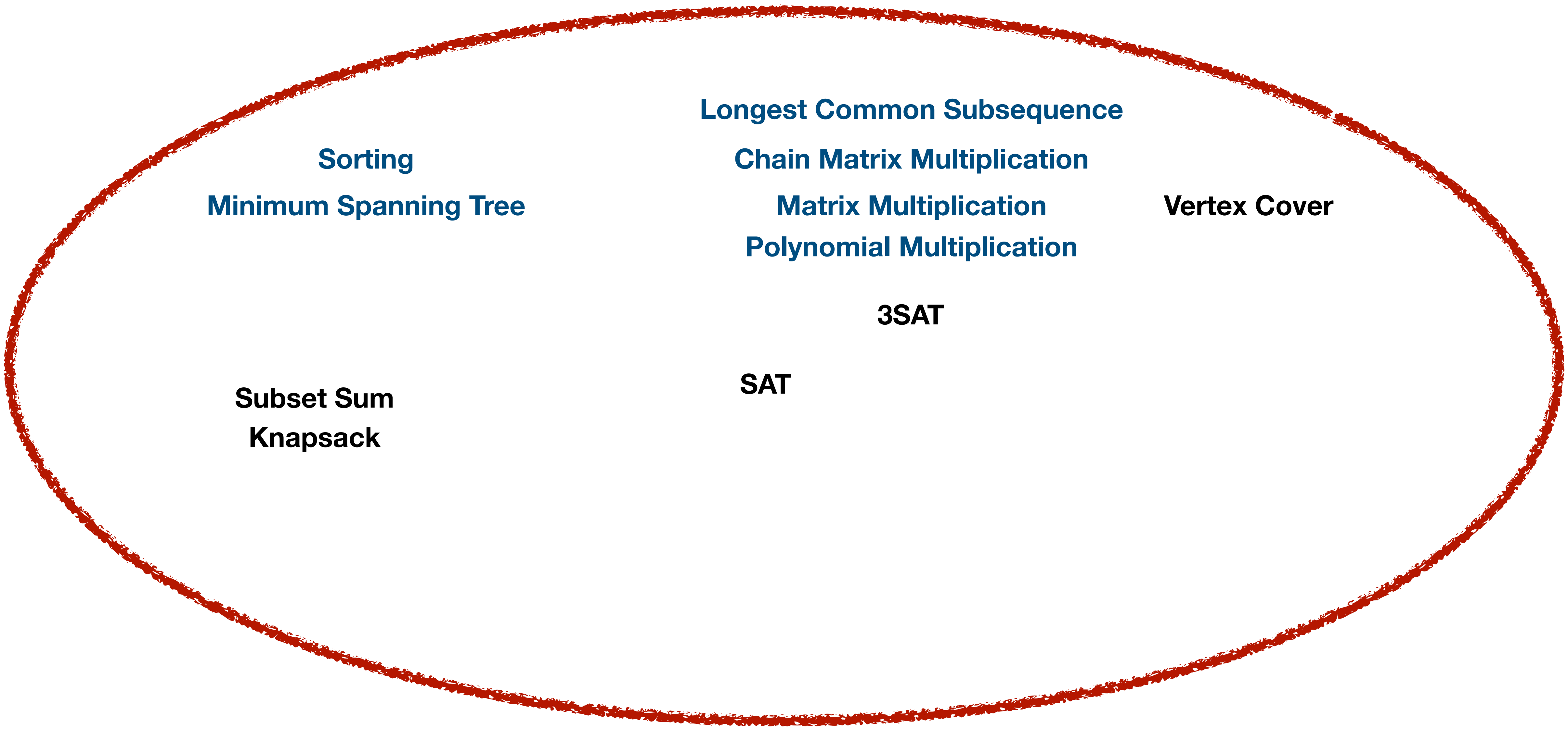

Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Polynomial Multiplication
SAT

The class NP

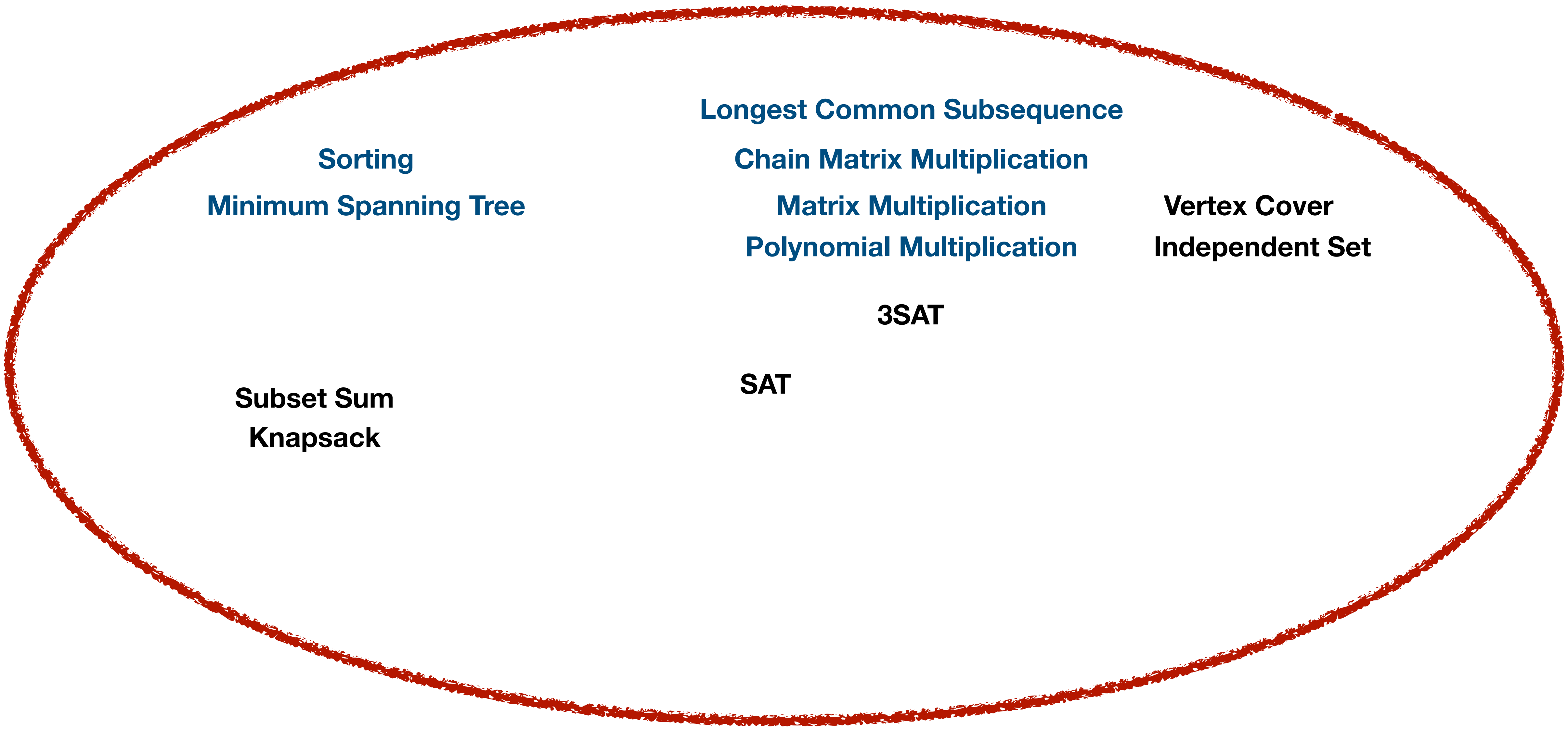


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
SAT
Subset Sum
Knapsack

The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
3SAT
SAT
Subset Sum
Knapsack

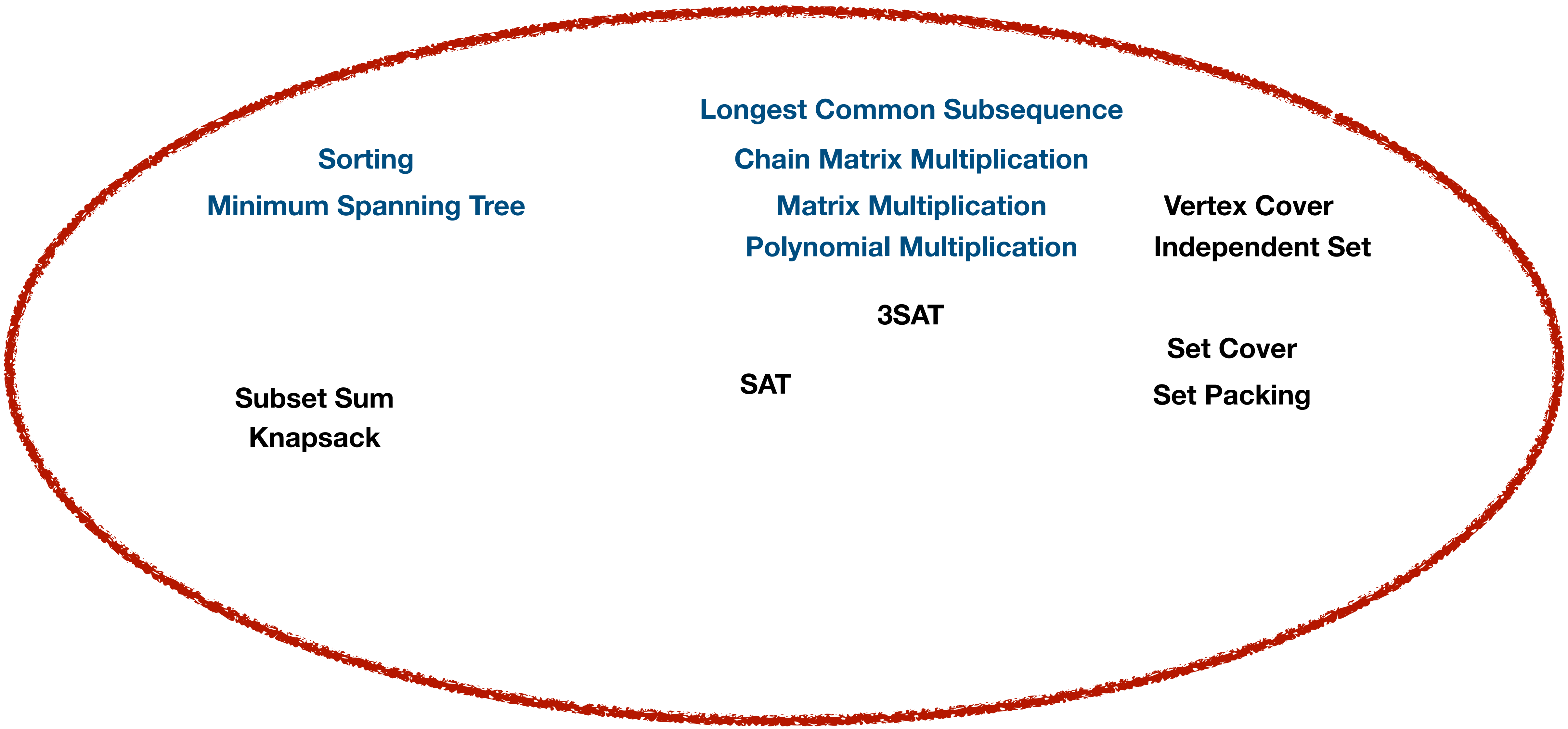
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
SAT
Subset Sum
Knapsack

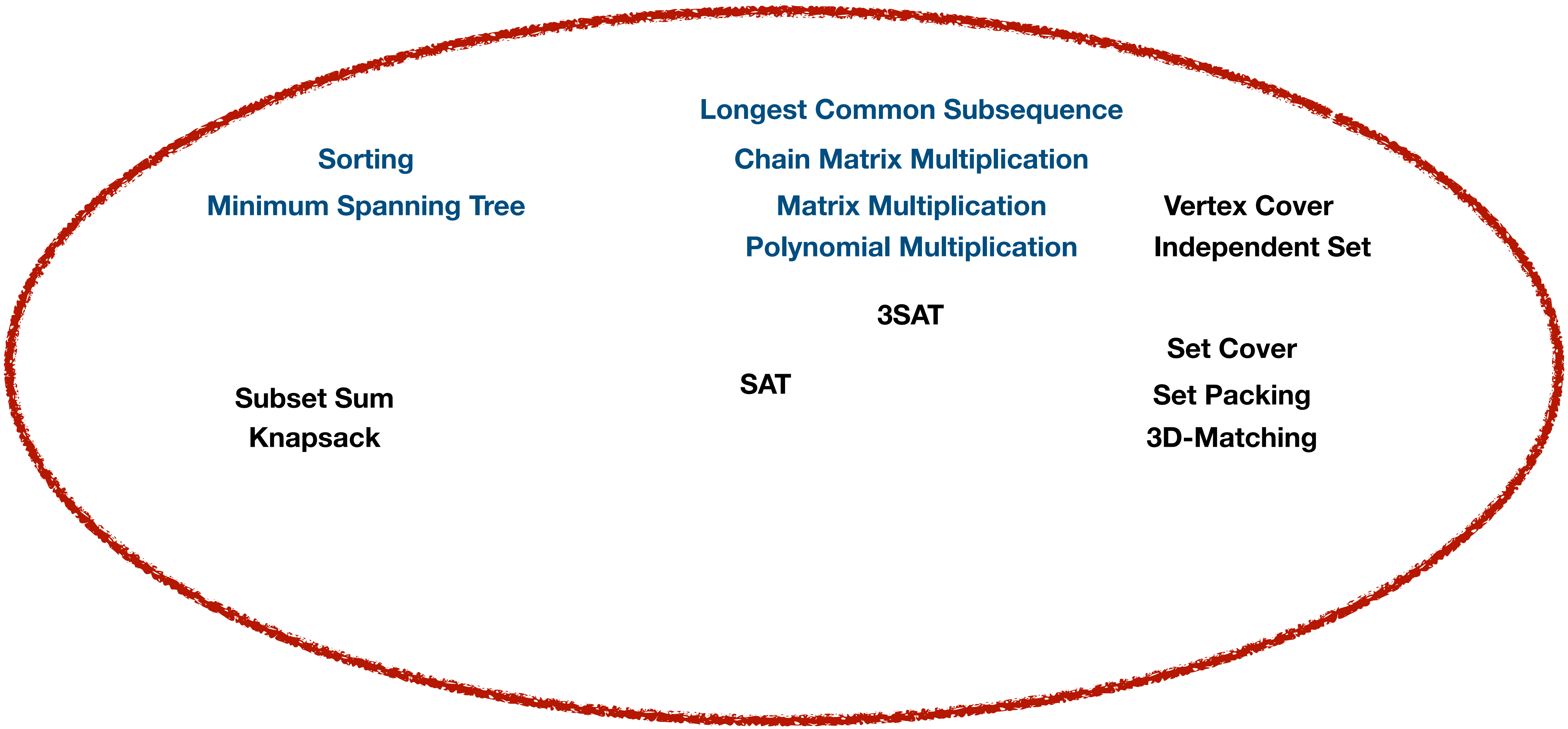
The class NP

Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
SAT
Subset Sum
Set Packing
Knapsack

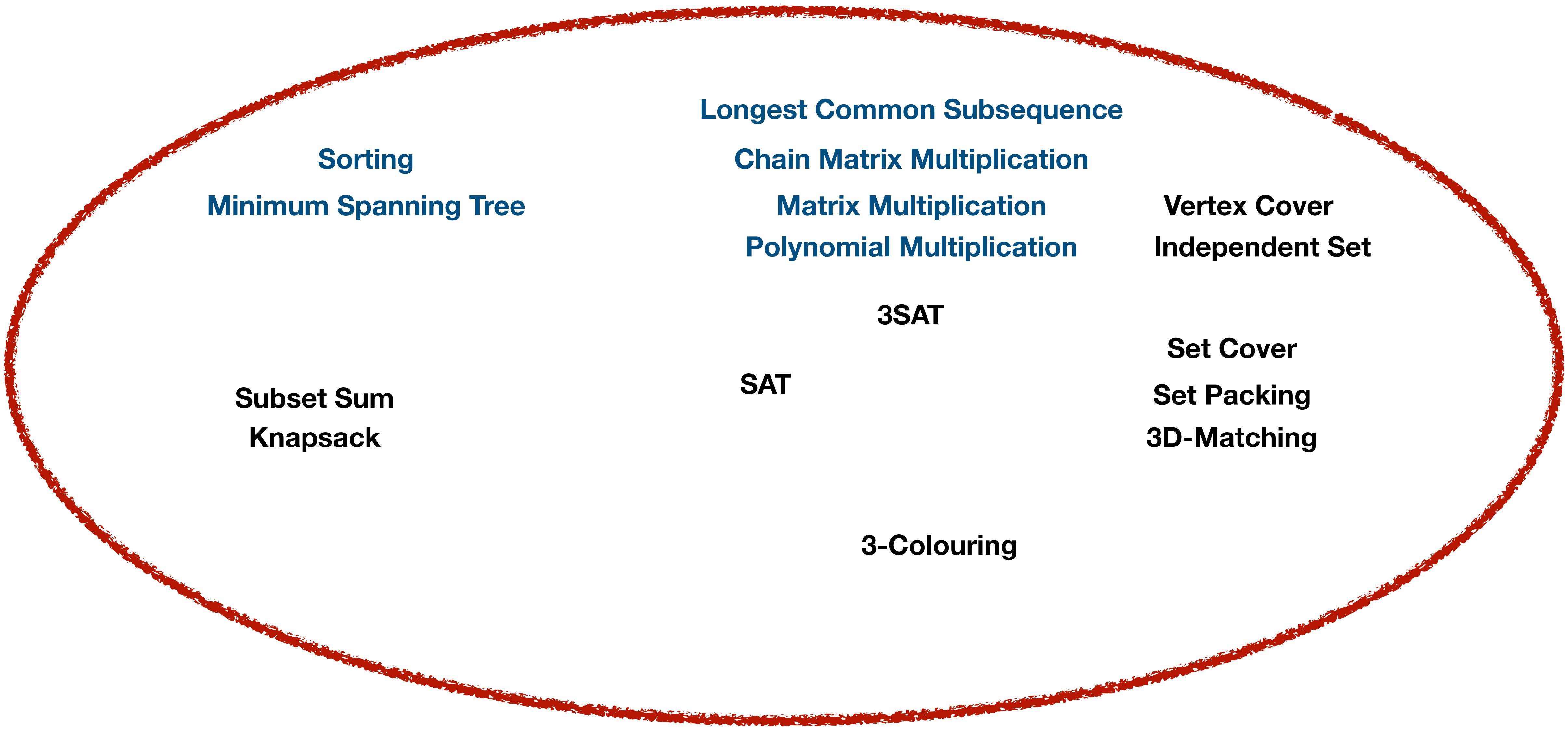
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
Set Cover
SAT
Subset Sum
Set Packing
Knapsack

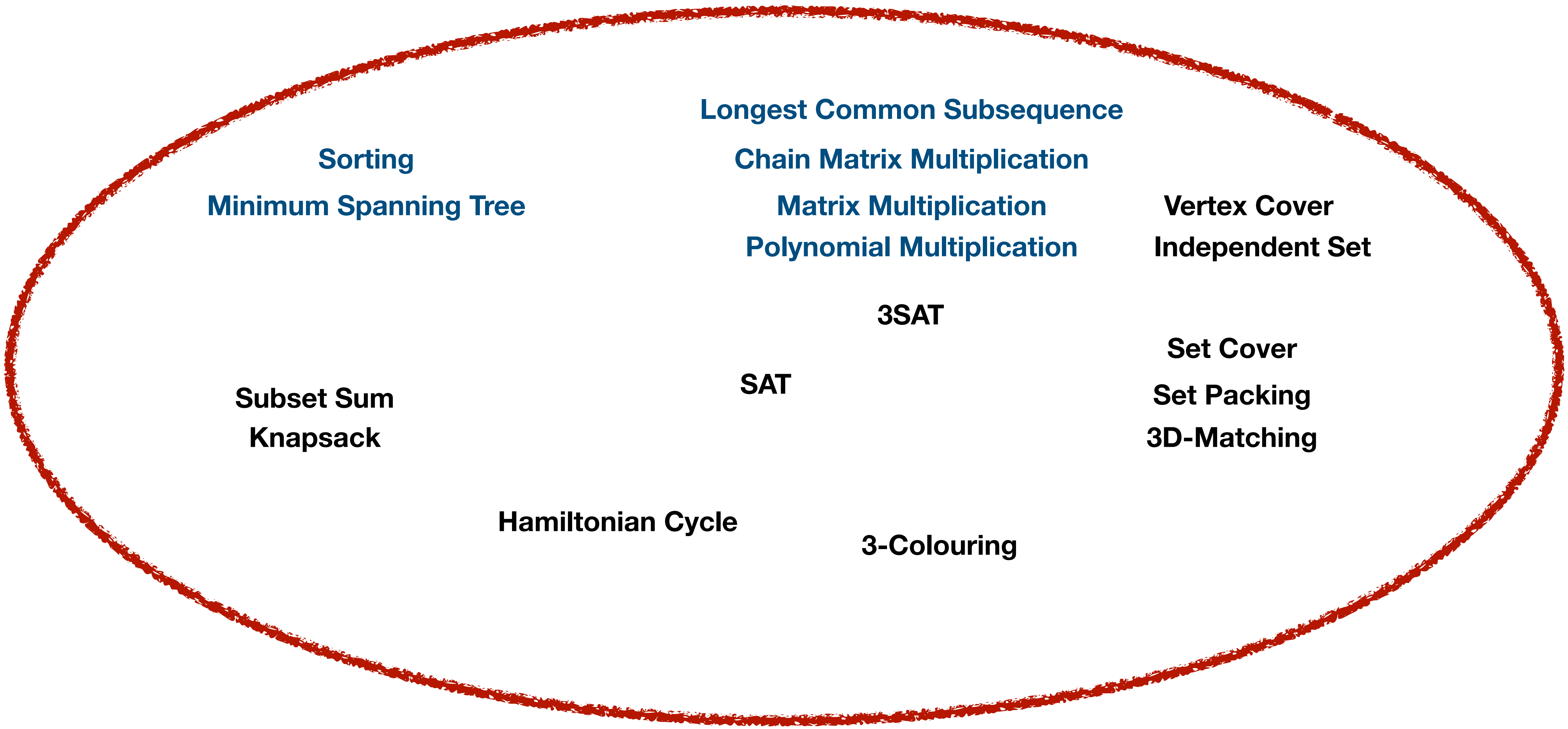
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
Set Cover
SAT
Subset Sum
Set Packing
Knapsack
3D-Matching

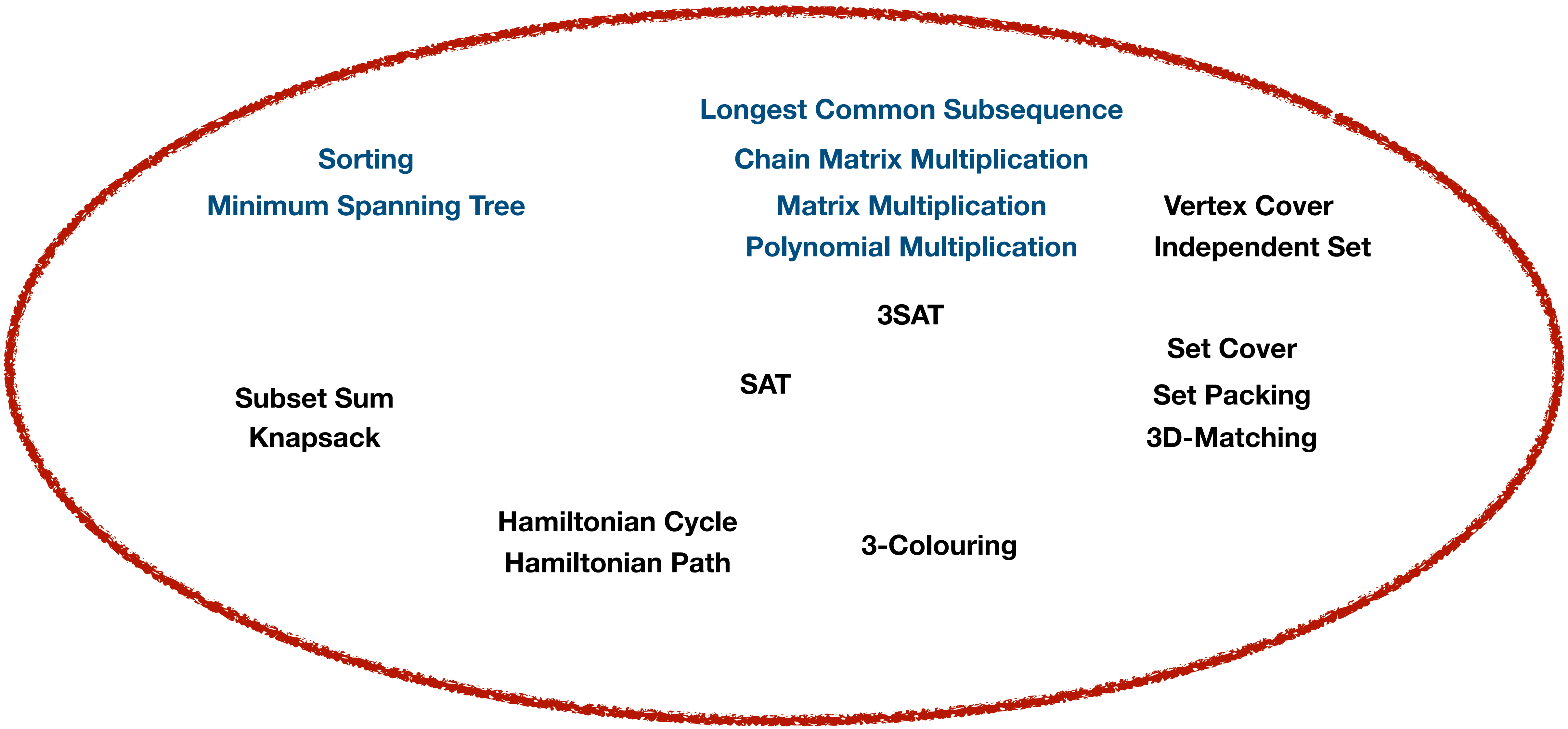
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
Set Cover
SAT
Subset Sum
Set Packing
Knapsack
3D-Matching
3-Colouring

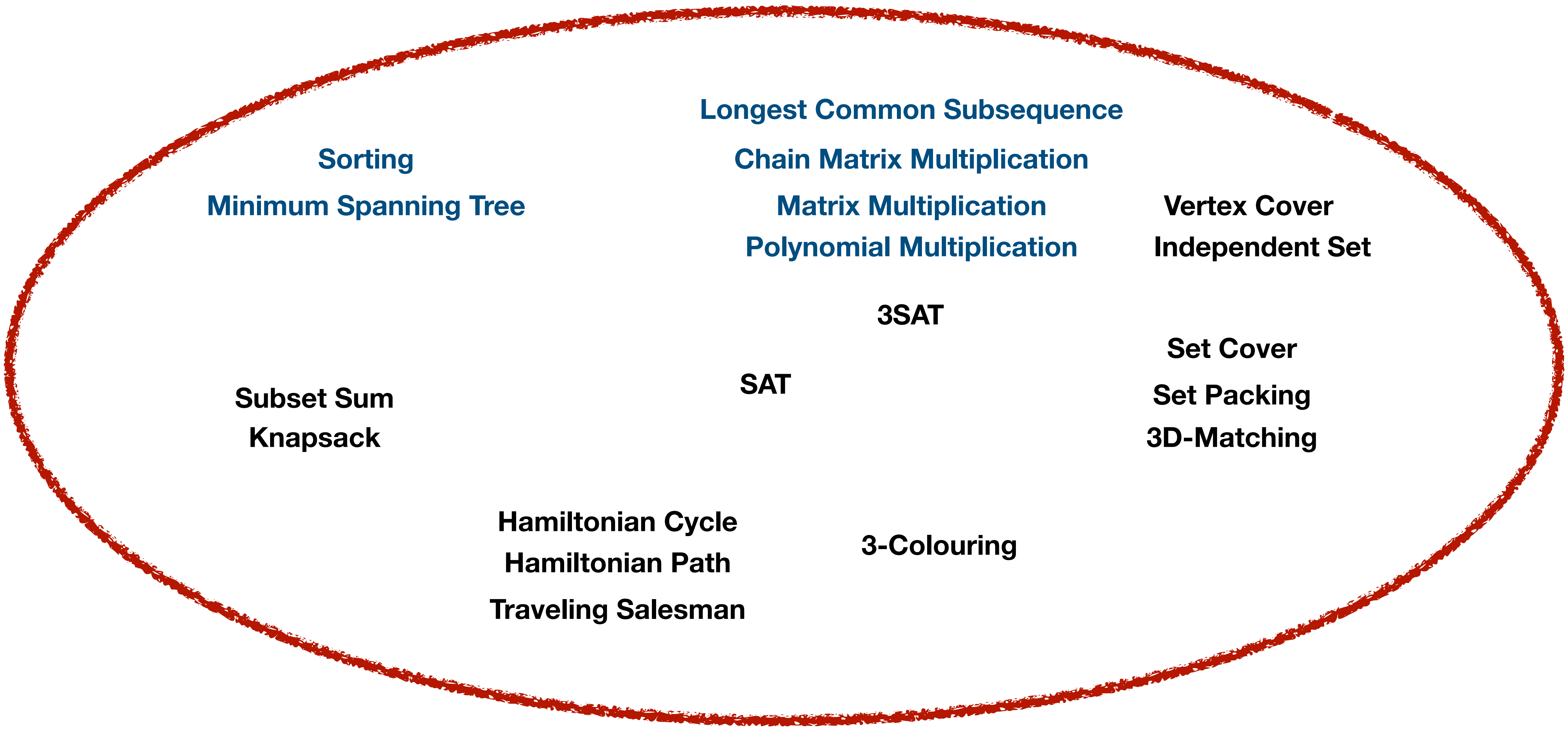
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
Set Cover
SAT
Subset Sum
Set Packing
Knapsack
3D-Matching
Hamiltonian Cycle
3-Colouring

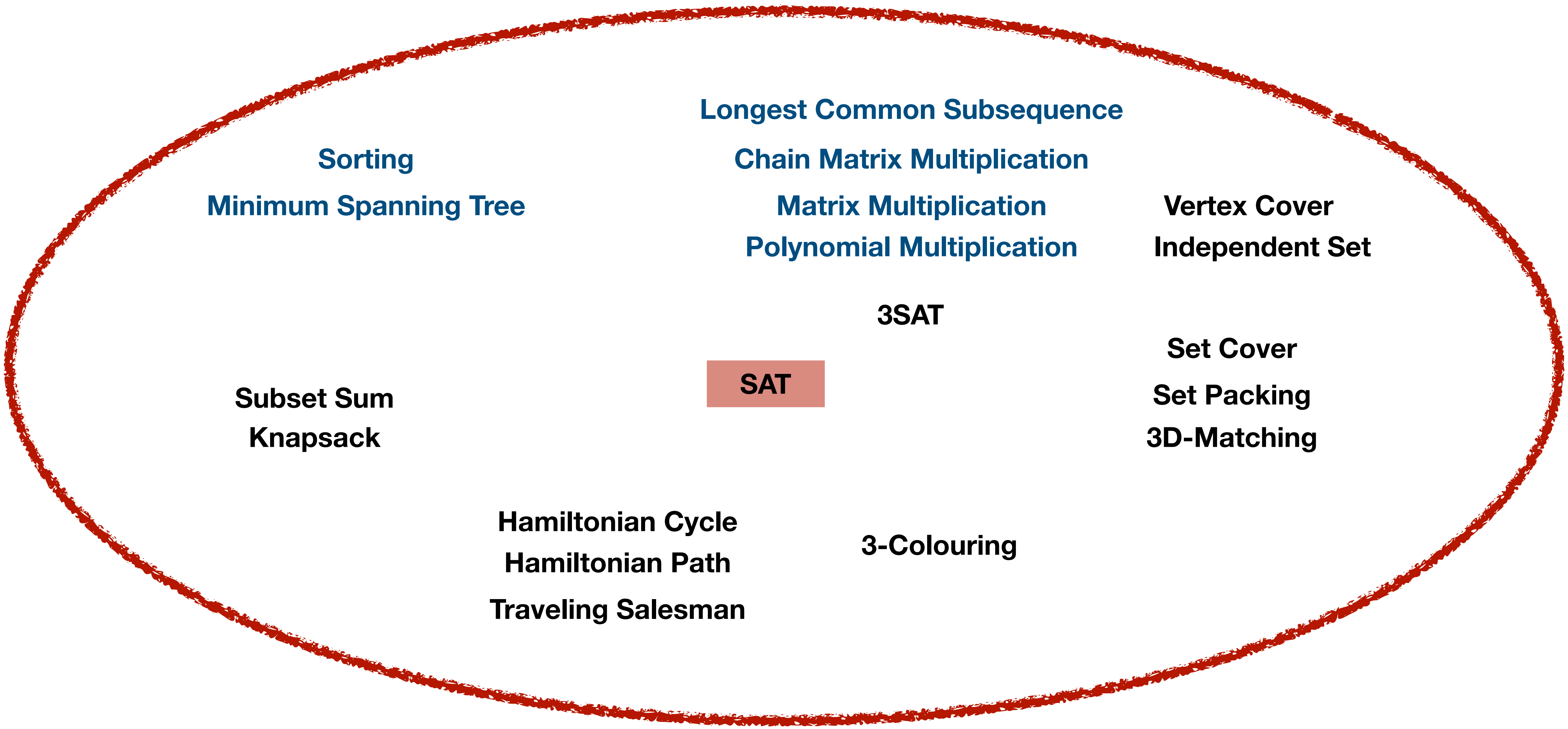
The class NP


Longest Common Subsequence
Chain Matrix Multiplication
Matrix Multiplication
Polynomial Multiplication
Sorting
Minimum Spanning Tree
Vertex Cover
Independent Set
3SAT
SAT
Set Cover
Set Packing
3D-Matching
Subset Sum
Knapsack
Hamiltonian Cycle
Hamiltonian Path
3-Colouring

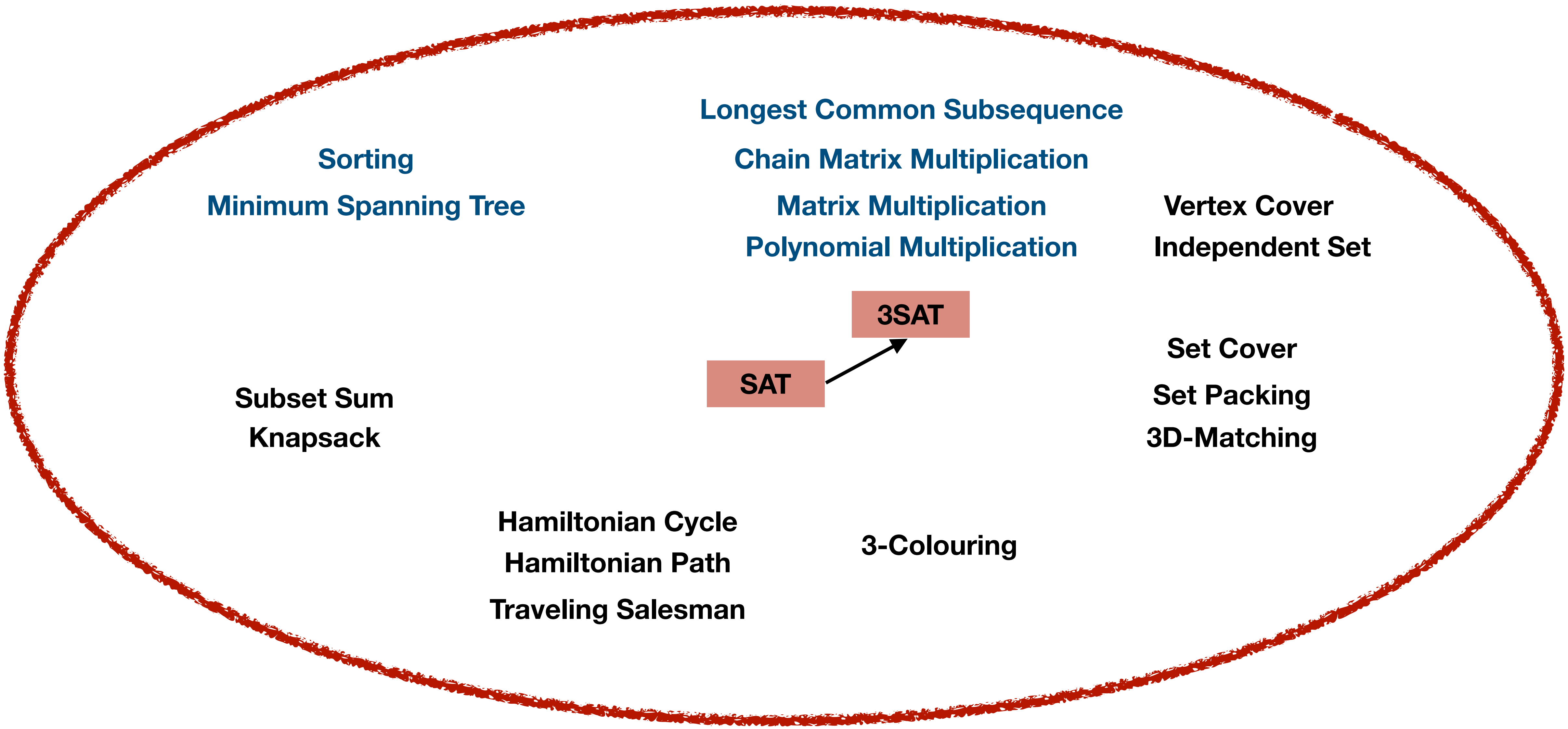
The class NP


Longest Common Subsequence
Sorting
Chain Matrix Multiplication
Minimum Spanning Tree
Matrix Multiplication
Vertex Cover
Polynomial Multiplication
Independent Set
3SAT
Set Cover
SAT
Subset Sum
Set Packing
Knapsack
3D-Matching
Hamiltonian Cycle
3-Colouring
Hamiltonian Path
Traveling Salesman

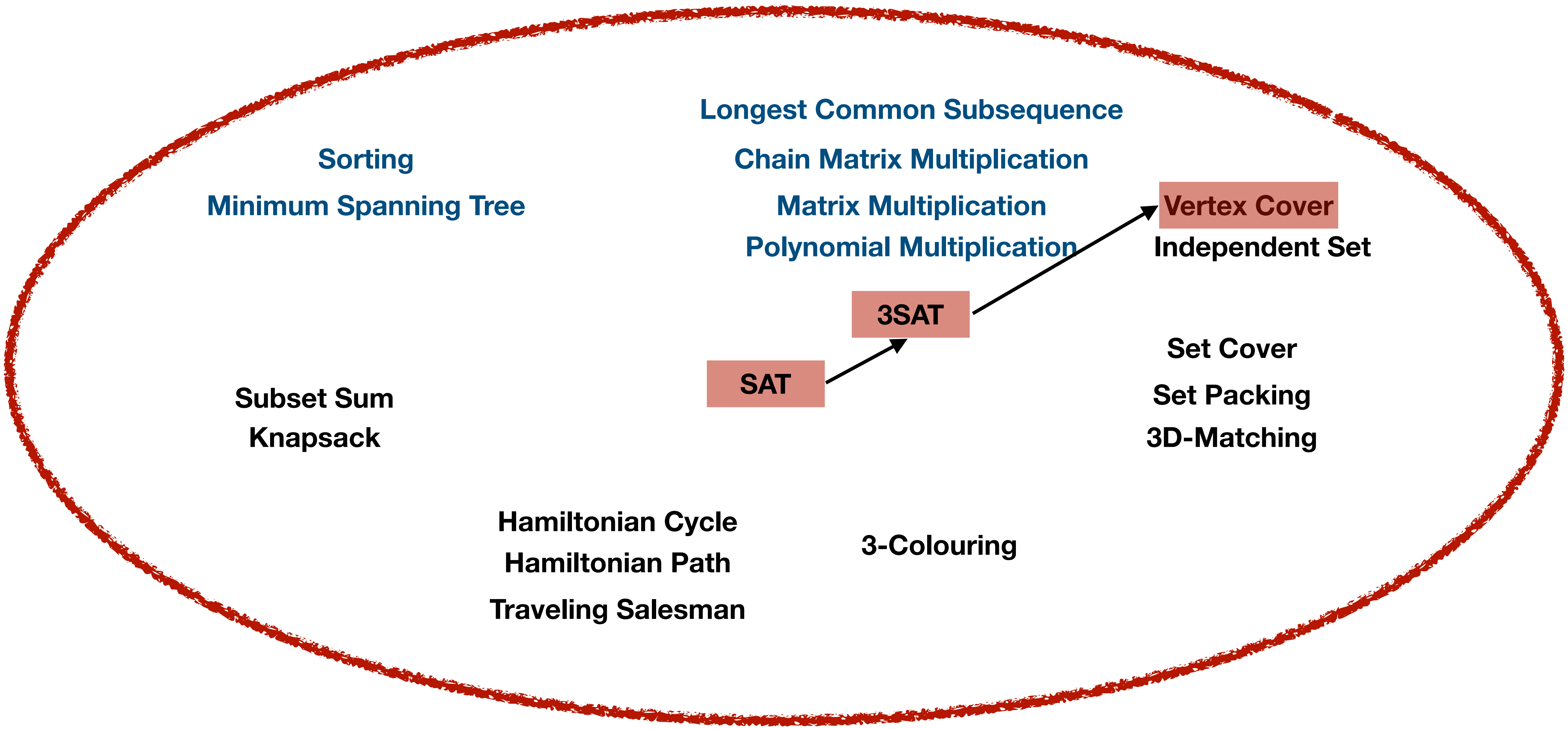
NP-completeness



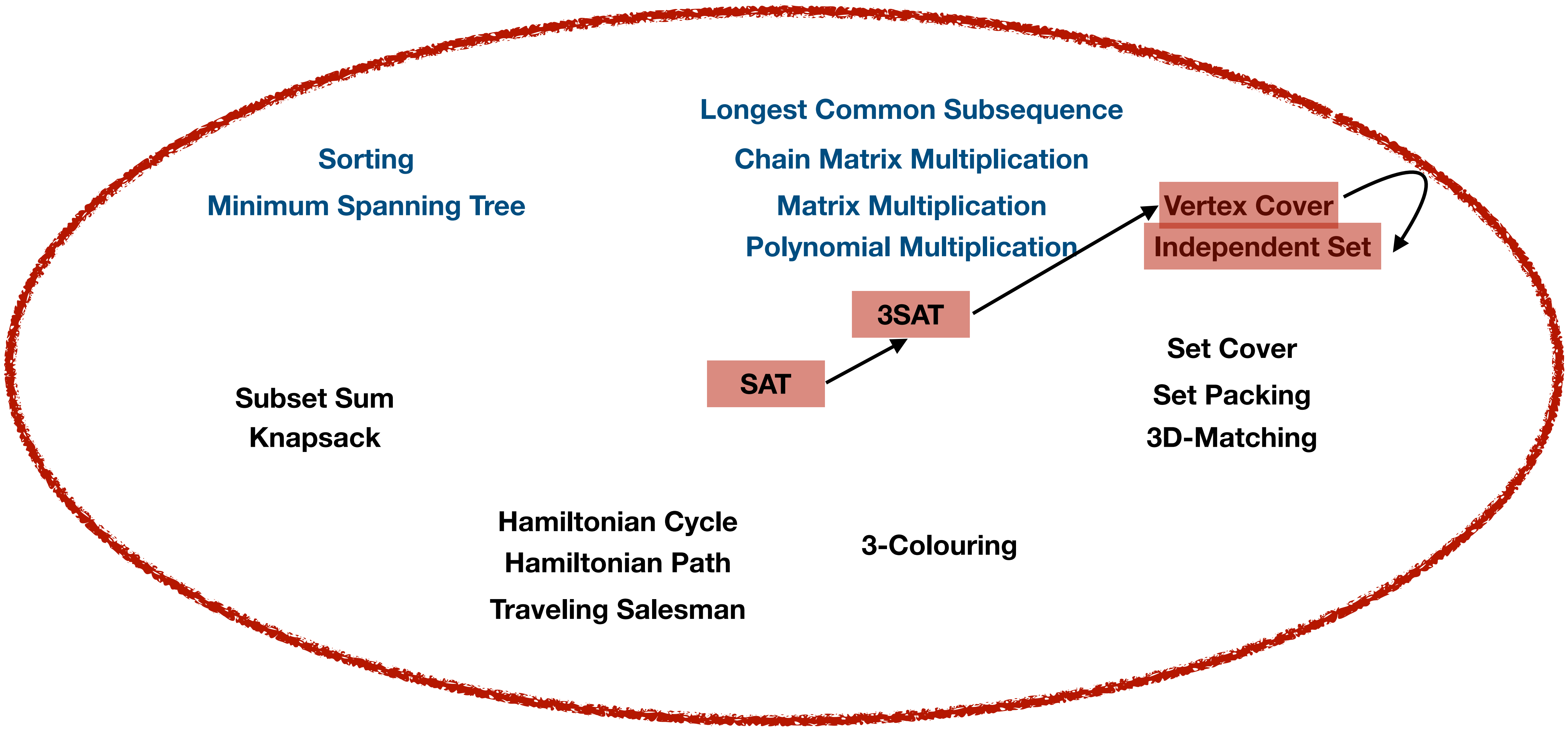
NP-completeness



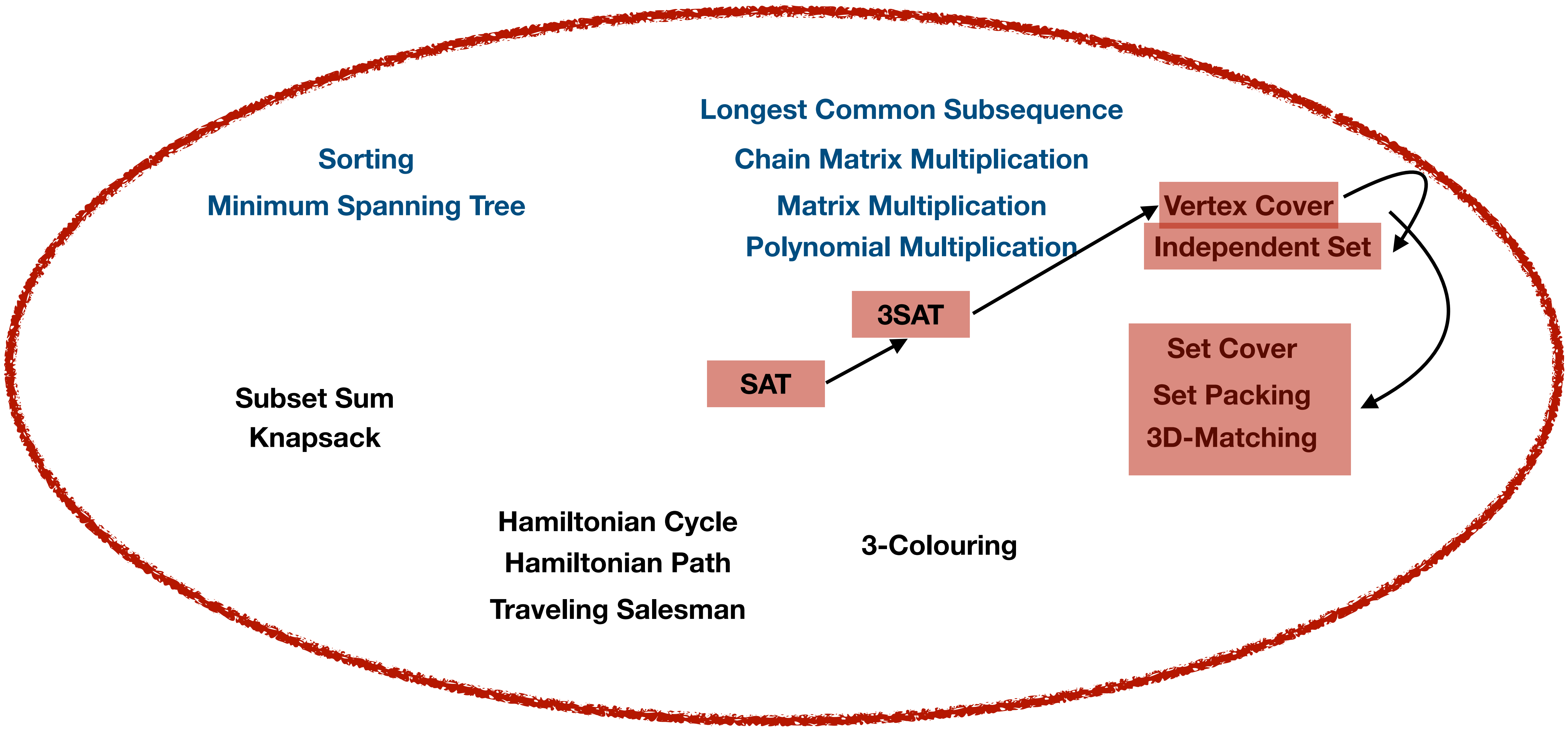
NP-completeness



NP-completeness



NP-completeness



Proving NP-completeness

Proving NP-completeness

Suppose that you are given a problem A and you want to prove that it is NP-complete.

Proving NP-completeness

Suppose that you are given a problem A and you want to prove that it is NP-complete.

First, prove that A is in NP.

Usually by observing that a solution is efficiently checkable.

Proving NP-completeness

Suppose that you are given a problem A and you want to prove that it is NP-complete.

First, prove that A is in NP.

Usually by observing that a solution is efficiently checkable.

Then prove that A is NP-hard.

Construct a polynomial time reduction from some NP-complete problem P.

Reduction strategies

Reduction strategies

The idea is to find a problem that looks similar to the one we are trying to prove NP-hardness for.

Reduction strategies

The idea is to find a problem that looks similar to the one we are trying to prove NP-hardness for.

Try to think of reductions you have seen in the past.

Reduction strategies

The idea is to find a problem that looks similar to the one we are trying to prove NP-hardness for.

Try to think of reductions you have seen in the past.

This takes time!

NP-completeness, a taxonomy

Packing problems

Independent Set
Set Packing

Covering problems

Vertex Cover
Set Cover

Partitioning problems

3D-Matching
Graph Colouring

Sequencing problems

Hamiltonian Cycle
Hamiltonian Path
Traveling Salesman

Numerical problems

Subset Sum
Knapsack

Constraint Satisfaction problems

3 SAT

The effect of NP-hardness

The effect of NP-hardness

Imagine that you have a new favourite problem P.

The effect of NP-hardness

Imagine that you have a new favourite problem P.

You try to design a polynomial-time algorithm for it but you find it hard to do so.

The effect of NP-hardness

Imagine that you have a new favourite problem P.

You try to design a polynomial-time algorithm for it but you find it hard to do so.

Then you discover that one of all of these NP-complete problems can be reduced to it.

The effect of NP-hardness

Imagine that you have a new favourite problem P.

You try to design a polynomial-time algorithm for it but you find it hard to do so.

Then you discover that one of all of these NP-complete problems can be reduced to it.

This means that if you succeeded in your quest, you would solve all of these problems in polynomial-time.

The effect of NP-hardness

Imagine that you have a new favourite problem P.

You try to design a polynomial-time algorithm for it but you find it hard to do so.

Then you discover that one of all of these NP-complete problems can be reduced to it.

This means that if you succeeded in your quest, you would solve all of these problems in polynomial-time.

That would mean that you are smarter than generations of researchers and pretty much anyone else that has studied computer science ever.

The effect of NP-hardness

Imagine that you have a new favourite problem P.

You try to design a polynomial-time algorithm for it but you find it hard to do so.

Then you discover that one of all of these NP-complete problems can be reduced to it.

This means that if you succeeded in your quest, you would solve all of these problems in polynomial-time.

That would mean that you are smarter than generations of researchers and pretty much anyone else that has studied computer science ever.

I don’t know about you, but I would probably be convinced that I am not going to come up with a polynomial-time algorithm!

My problem is NP-hard, what can I do?

My problem is NP-hard, what can I do?

Don’t give up hope just yet:

My problem is NP-hard, what can I do?

<u>Don't give up hope just yet:</u>

NP-hardness is a worst-case impossibility.

My problem is NP-hard, what can I do?

Don’t give up hope just yet:

NP-hardness is a worst-case impossibility.

The problem might be efficiently solvable in the average case, or on typical instances encountered in practice.

My problem is NP-hard, what can I do?

Don't give up hope just yet:

NP-hardness is a worst-case impossibility.

The problem might be efficiently solvable in the average case, or on typical instances encountered in practice.

I can use heuristics or approximation algorithms that don't solve the problem *exactly*, but *approximately*.

My problem is NP-hard, what can I do?

Don’t give up hope just yet:

NP-hardness is a worst-case impossibility.

The problem might be efficiently solvable in the average case, or on typical instances encountered in practice.

I can use heuristics or approximation algorithms that don’t solve the problem *exactly*, but *approximately*.

I can formulate the problem as an ILP and ask my clever solver software to solve it.

My problem is NP-hard, what can I do?

<u>Don’t give up hope just yet:</u>

If you would like to know more, talk to your local lecturer.

NP-hardness is a worst-case impossibility.

The problem might be efficiently solvable in the average case, or on typical instances encountered in practice.

I can use heuristics or approximation algorithms that don’t solve the problem *exactly*, but *approximately*.

I can formulate the problem as an ILP and ask my clever solver software to solve it.