NP-Completeness

CPSC 6109 - Algorithms Analysis and Design

Dr. Hyrum D. Carroll

April 10, 2024

Complexity Classes

Classes:

- **P**: Problems that are solvable in polynomial time: $O(n^k)$
- **NP**: Verifiable in polynomial time
  (verifiable means we can check the answer)

All problems in P are in NP: $P \subseteq NP$
(because we can more than check an answer, we can solve it in polynomial time)
The open question is if it's $P \subset NP$

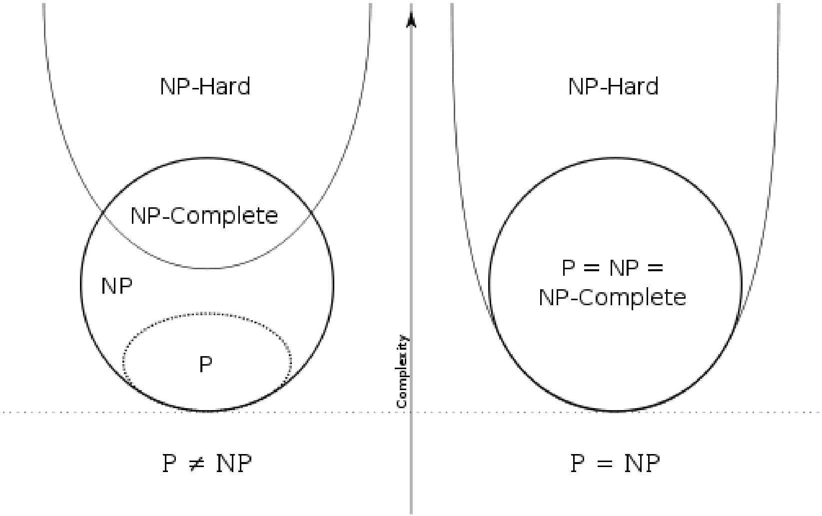
Complexity Classes

Classes:

- **P**: Problems that are solvable in polynomial time: $O(n^k)$
- **NP**: Verifiable in polynomial time
  (verifiable means we can check the answer)
- **NP-Complete**: as hard as any other problem in NP

All problems in P are in NP: $P \subseteq NP$
(because we can more than check an answer, we can solve it in polynomial time)
The open question is if it's $P \subset NP$

Complexity Classes (illustrated)



Source: Wikimedia Commons, user: Behnam Esfahbod

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?
- Formally, we would phrase this as, given graph G and vertices u and v and a number k, is there a simple path from u to v with at most k edges?
- Is this problem in P or NP?

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?
- Formally, we would phrase this as, given graph G and vertices u and v and a number k, is there a simple path from u to v with at most k edges?
- Is this problem in P or NP?
- Given a solution, can we determine if it's acyclical and has at most k edges in polynomial time?

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?
- Formally, we would phrase this as, given graph G and vertices u and v and a number k, is there a simple path from u to v with at most k edges?
- Is this problem in P or NP?
- Given a solution, can we determine if it's acyclical and has at most k edges in polynomial time?
- Yes, so this problem is in NP

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?
- Formally, we would phrase this as, given graph G and vertices u and v and a number k, is there a simple path from u to v with at most k edges?
- Is this problem in P or NP?
- Given a solution, can we determine if it's acyclical and has at most k edges in polynomial time?
- Yes, so this problem is in NP
- Is it in P?

Example: Long Simple Paths

- Simple paths are acyclic
- Is determining if a path is simple in P or NP?
- Formally, we would phrase this as, given graph G and vertices u and v and a number k, is there a simple path from u to v with at most k edges?
- Is this problem in P or NP?
- Given a solution, can we determine if it's acyclical and has at most k edges in polynomial time?
- Yes, so this problem is in NP
- Is it in P?
- Can we develop an algorithm that runs in polynomial time?

Exercise

- Is determining the solution to a linear programming problem in P or NP?

Exercise: Solution

- Is determining the solution to a linear programming problem in P, NP or NP-Complete?
- Cast the question as a yes-no question:
- To determine if it is in NP, can we, in polynomial-time, determine if a solution is correct?

Exercise: Solution

- Is determining the solution to a linear programming problem in P, NP or NP-Complete?
- Cast the question as a yes-no question:
- To determine if it is in NP, can we, in polynomial-time, determine if a solution is correct?
- Yes

Exercise: Solution

- Is determining the solution to a linear programming problem in P, NP or NP-Complete?
- Cast the question as a yes-no question:
- To determine if it is in NP, can we, in polynomial-time, determine if a solution is correct?
- Yes
- To determine if it is in P, can we, in polynomial-time, calculate a solution?

Exercise: Solution

- Is determining the solution to a linear programming problem in P, NP or NP-Complete?
- Cast the question as a yes-no question:
- To determine if it is in NP, can we, in polynomial-time, determine if a solution is correct?
- Yes
- To determine if it is in P, can we, in polynomial-time, calculate a solution?
- Yes
- So, in P

Exercise: Solution

- Is determining the solution to a linear programming problem in P, NP or NP-Complete?
- Cast the question as a yes-no question:
- To determine if it is in NP, can we, in polynomial-time, determine if a solution is correct?
- Yes
- To determine if it is in P, can we, in polynomial-time, calculate a solution?
- Yes
- So, in P
- But integer linear programming is NP-Complete :)

How knowing about complexity can help you

- If you're asked to implement a solution to a problem that is NP-Complete, don't waste your time coming up with an exact solution, but focus on:
  - Approximations (Chapter 35)
  - Heuristics
  - Accepting that an exponential run-time is the best you can do
  - Determine if you can solve just a subset of the problems efficiently

Showing a Problem is NP-Complete

- Instead of how easy a problem is, we're saying, **how hard the problem is**
- Instead of proving an efficient algorithm, we showing that, **no efficient algorithm is likely to exist**

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output
  - Example: `Shortest Paths` (given a graph and weights, what's the shortest path between vertices u and v)

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output
  - Example: `Shortest Paths` (given a graph and weights, what's the shortest path between vertices u and v)
- NP-Completeness applies to decision problems

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output
  - Example: `Shortest Paths` (given a graph and weights, what's the shortest path between vertices u and v)
- NP-Completeness applies to decision problems (yes / no problems)

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output
  - Example: `Shortest Paths` (given a graph and weights, what's the shortest path between vertices u and v)
- NP-Completeness applies to decision problems (yes / no problems)
- Usually we can just bound an optimization problem to make it a decision problem

Decision Problems vs. Optimization Problems

- Usually a problem is an optimization problem:
  - For each input, what's the best output
  - Example: `Shortest Paths` (given a graph and weights, what's the shortest path between vertices u and v)
- NP-Completeness applies to decision problems (yes / no problems)
- Usually we can just bound an optimization problem to make it a decision problem
- Example:
  - `Shortest Paths` $\rightarrow$ `Path`
  - Given a graph and weights **and threshold** k, is there a path between vertices u and v **that has at most** k **edges**

Decision Problems vs. Optimization Problems (cont'd)

- Makes problems easier (or at least no harder) than the optimization problem

Decision Problems vs. Optimization Problems (cont'd)

- Makes problems easier (or at least no harder) than the optimization problem
- Often, solving the optimization problem will solve the decision problem (because it's a subset of it)

Decision Problems vs. Optimization Problems (cont'd)

- Makes problems easier (or at least no harder) than the optimization problem
- Often, solving the optimization problem will solve the decision problem (because it's a subset of it)
- So, the decision problem version is easier

Decision Problems vs. Optimization Problems (cont'd)

- Makes problems easier (or at least no harder) than the optimization problem
- Often, solving the optimization problem will solve the decision problem (because it's a subset of it)
- So, the decision problem version is easier
- If we can prove that the decision problem is hard, then we can prove that the optimization problem is hard

Reductions

- Almost every NP-Complete proof makes a reduction of one problem to another

Polynomial-Time Reductions

- **Instance**: 1 particular set of inputs

Polynomial-Time Reductions

- **Instance**: 1 particular set of inputs
- We want to solve a decision problem A in polynomial-time

Polynomial-Time Reductions

- **Instance**: 1 particular set of inputs
- We want to solve a decision problem A in polynomial-time
- We have a known polynomial-time solution to decision problem B

Polynomial-Time Reductions

- **Instance**: 1 particular set of inputs
- We want to solve a decision problem A in polynomial-time
- We have a known polynomial-time solution to decision problem B
- We have a polynomial-time mapping for every instant of A (α) to an instance of B (β) such that the answer to α is yes if and only if the answer to β is yes

Polynomial-Time Reductions

- **Instance**: 1 particular set of inputs
- We want to solve a decision problem A in polynomial-time
- We have a known polynomial-time solution to decision problem B
- We have a polynomial-time mapping for every instant of A (α) to an instance of B (β) such that the answer to α is yes if and only if the answer to β is yes

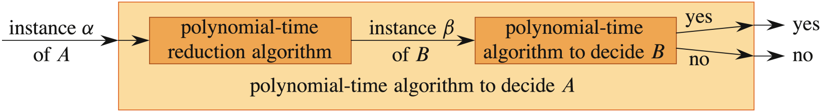


Figure 34.1 from Introduction to Algorithms 4^{th} Edition

Polynomial-Time Reductions (Pseudocode)

```
Boolean B( β ); // known solution

Boolean A( α ){
    return B( transformArgs( α ) );
}
```

- If `B()` and `transformArgs()` each take polynomial-time, then `A()` takes polynomial-time

Polynomial-Time Reductions (Pseudocode)

```
Boolean B( β ); // known solution

Boolean A( α ){
    return B( transformArgs( α ) );
}
```

- If `B()` and `transformArgs()` each take polynomial-time, then `A()` takes polynomial-time
- We "reduce" problem A to solving problem B

Polynomial-Time Reductions (Pseudocode)

```
Boolean B( β ); // known solution

Boolean A( α ){
    return B( transformArgs( α ) );
}
```

- If `B()` and `transformArgs()` each take polynomial-time, then `A()` takes polynomial-time
- We "reduce" problem A to solving problem B
- We use the easiness of B to prove the easiness of A

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:
  1. A problem A that does not have a polynomial-time solution

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:
  1. A problem A that does not have a polynomial-time solution
  2. A polynomial-time mapping of every instance of A to an instance of B

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:
  1. A problem A that does not have a polynomial-time solution
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that no polynomial-time solution can exist for B:

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:
  1. A problem A that does not have a polynomial-time solution
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that no polynomial-time solution can exist for B:
  - Assume that B has a polynomial-time solution. Then, we can solve all instances of A using B (using polynomial-time reductions)
  - That's not possible, so B cannot have a polynomial-time solution

Polynomial-Time Reductions In Reverse

- For NP-Complete, we want to show at least how hard a problem is
- Assume we have:
  1. A problem A that does not have a polynomial-time solution
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that no polynomial-time solution can exist for B:
  - Assume that B has a polynomial-time solution. Then, we can solve all instances of A using B (using polynomial-time reductions)
  - That's not possible, so B cannot have a polynomial-time solution

Polynomial-Time Reductions In Reverse $\rightarrow$ NP-Complete

For NP-Completeness:

Polynomial-Time Reductions In Reverse $\rightarrow$ NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)

Polynomial-Time Reductions In Reverse $\rightarrow$ NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)
- Assume we have:

Polynomial-Time Reductions In Reverse → NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)
- Assume we have:
  1. A problem A that is NP-Complete

Polynomial-Time Reductions In Reverse $\rightarrow$ NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)
- Assume we have:
  1. A problem A that is NP-Complete
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that B is NP-Complete.

Polynomial-Time Reductions In Reverse → NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)
- Assume we have:
  1. A problem A that is NP-Complete
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that B is NP-Complete.
  - Assume that B has a polynomial-time solution. Then, we can solve all instances of A using B (using polynomial-time reductions).

Polynomial-Time Reductions In Reverse $\rightarrow$ NP-Complete

For NP-Completeness:

- Use polynomial-time reductions in reverse to show that problem B is NP-Complete (if A is NP-Complete)
- Assume we have:
  1. A problem A that is NP-Complete
  2. A polynomial-time mapping of every instance of A to an instance of B
- Using proof by contradiction, we can show that B is NP-Complete.
  - Assume that B has a polynomial-time solution. Then, we can solve all instances of A using B (using polynomial-time reductions).
  - That's not possible, so B cannot have a polynomial-time solution.

Exercise: One NP-Complete Problem

- Choose one of Karp's 21 NP-Complete problems
- Describe your algorithm
- Describe how we know it's NP-Complete

Exercise: Heirarchy of Karp's 21 NP-Complete Problems

- Draw a heirarchy (showing which problem was reduce to a known NP-complete problem) for each one of Karp's 21 NP-complete problems