

Advalgo quick Summary



Gabriel rovesti

Summary

[2 Exam – First Part 3](#_Toc167717225)

[2.1 Complexities and NP-Hard problems 3](#_Toc167717226)

[2.1.1 Algorithms 3](#_Toc167717227)

[2.1.2 NP-Hard Problems 3](#_Toc167717228)

[3 Exam – Second Part 4](#_Toc167717229)

[3.1 How to do reductions 4](#_Toc167717230)

[3.1.1 Complete Example 5](#_Toc167717231)

[3.1.2 Known reductions seen inside the course 6](#_Toc167717232)

[3.2 How to do approximation algorithms 7](#_Toc167717233)

[3.2.1 Known approximation algorithms seen in course 8](#_Toc167717234)

# Exam – First Part

## Complexities and NP-Hard problems

### Algorithms

* + DFS/BFS =
    - Also called in exams “Graph connectivity”
  + Minimum Spanning Tree (MST)
    - Prim’s algorithm =
      * Prim with heaps =
    - Kruskal’s algorithm =
      * Kruskal with Union-Find = = best algorithm
  + Single-source shortest paths (SSSP)
    - Dijkstra’s algorithm =
      * Dijkstra with heaps= = best algorithm
    - Bellman-Ford’s algorithm =
  + All-pairs shortest paths (APSP)
    - Bellman-Ford with dynamic programming =
    - Floyd-Warshall =
  + Maximum flow
    - Ford-Fulkerson =

### NP-Hard Problems

* NP-Hard problems (seen in the course)
  + TSP – Traveling Salesperson Problem
  + Metric TSP
  + Maximum Independent Set (or Maximum independent set)
  + Vertex cover (or Minimum Vertex Cover)
  + 3SAT
  + Hamiltonian circuit
  + Clique (or Maximum Clique)
  + Set Cover

# Exam – Second Part

## How to do reductions

Here's a general structure you can follow when solving a reduction problem to prove that a problem is NP-hard by reducing a known NP-hard problem to :

1. Introduction

* Goal: To prove that problem is NP-hard by reducing a known NP-hard problem to .

2. Problem Definition

* Define problem Y (the known NP-hard problem)
  + Specify the input format and the desired output
* Define problem X (the problem you want to prove is NP-hard)
  + Specify the input format and the desired output

3. Reduction process

* Given an instance of problem , construct an instance of problem
* Specify the steps to transform an instance of into an instance of
* Explain how the input of Y is used to create the input of
* Define any additional variables or structures needed for the reduction

4. Correctness proof

* If there is a solution to the instance of
  + then there is a corresponding solution to the constructed instance of
* If there is a solution to the constructed instance of
  + then there is a corresponding solution to the instance of

5. Polynomial-time reduction

* Argue that the reduction can be performed in polynomial time, in terms of size and time

### Complete Example

Input:

* For Ham, the input is a graph , where is the set of vertices and is the set of edges.
* For TSP, the input is a complete weighted graph where is the set of vertices, E' is the set of edges, and w is a weight function assigning a non-negative weight to each edge.

Output:

* For , the output is "Yes" if the graph G contains a Hamiltonian cycle (a cycle that visits each vertex exactly once) and "No" otherwise.
* For , the output is the minimum total weight of a Hamiltonian cycle in the graph G'.

Reduction:

Given an instance of (a graph ), we construct an instance of (a complete weighted graph ) as follows:

1. Set , so the vertices of are the same as the vertices of .

2. For each edge , set the weight in .

3. For each pair of vertices , set the weight in .

Now, we have an instance of TSP (the complete weighted graph ). We claim that has a Hamiltonian cycle if and only if the minimum total weight of a Hamiltonian cycle in is exactly .

To show the reduction is correct, we prove the following:

1. If has a Hamiltonian cycle, then there exists a Hamiltonian cycle in with a total weight of |V|.

* Suppose has a Hamiltonian cycle. This means there is a cycle that visits each vertex exactly once using only the edges in .
* In the constructed graph , the edges from the Hamiltonian cycle in have a weight of 1, and there are such edges.
* Thus, the total weight of this Hamiltonian cycle in is exactly

2. If there exists a Hamiltonian cycle in with a total weight of , then G has a Hamiltonian cycle.

* Suppose there is a Hamiltonian cycle in with a total weight of .
* Since the minimum weight of any edge in is 1, the Hamiltonian cycle in G' must use only edges with weight .
* By construction, the edges with weight in correspond to the edges in .
* Therefore, the Hamiltonian cycle in corresponds to a Hamiltonian cycle in .

This reduction shows that if we can solve efficiently, we can also solve efficiently. We construct an instance of from an instance of Ham, solve , and then interpret the solution to determine if the original graph has a Hamiltonian cycle.

### Known reductions seen inside the course

These are collected here just to clearly see them:

## How to do approximation algorithms

*Definition*: Let be an optimization problem and let be an algorithm for that returns, (in other words, the choice the algorithm makes). We say that has an approximation factor of if such that we have (for each one, the concrete translation in problems):

1. minimization problem (basically, an explicit *lower-bound* of the optimal solution)

Here I call the choice of the algorithm, say it’s Vertex Cover.

So, in a logic of a X-approximation algorithm :

a) Upper bound to the cost of (which is our solution, greedy choice made by us)

* The upper bound means instances for which the algorithm does not improve
* It is the worst-case performance of the algorithm compared to the optimal solution

b) Lower bound to the cost of (which is the optimal solution, selected by the algorithm)

* The lower bound means a value the algorithm is guaranteed to be greater or equal to, hence the smallest possible cost case (base case)
* Represents the best-case performance of the algorithm compared to the optimal solution

1. maximization problem (basically, an explicit *upper-bound* of the optimal solution)

So, in a logic of a X-approximation algorithm :

a) Upper bound to the cost of (which is the optimal solution, selected by the algorithm)

* Same observations as before

b) Lower bound to the cost of (which is our heuristic solution, greedy choice made by us)

* Same observations as before

Most common case is the *2-approximation algorithm*. What does this even mean? It’s an algorithm which returns a solution whose cost is at most *twice* the optimal. Specifically:

* it gives solutions that never cost more than twice that of optimal if it is a minimization problem
* or never provide less than half the optimal value if it is a maximization problem

So, it’s something in line of (given the structure above):

Sometimes, you will be asked to show an approximation is *tight*:

* that depends on your definition of approximation ratio
* normally the approximation ratio is defined as the worst ratio between optimal solution and the one produced by your algorithm
* if this is the case, all you need to show that the ratio is tight is come up with one bad example, which shows it works for all sizes

### Known approximation algorithms seen in course

These are collected here just to clearly see them (note: they are all minimization problems):

* 2-approximation algorithm for Vertex Cover
* 2-approximation algorithm for Metric TSP (where

* 1.5 approximation algorithm for Metric TSP
* logarithmic algorithm () for Set Cover