NP Complete Project

CS 452 - Fall 2025

Learning Objectives

- learn about a specific NP-Complete problem and its properties
- Create multiple approximation methods citing each method's strengths and weaknesses
- Implement a reduction to another NP-Complete problem and characterize any performance differences
- Use AI technologies to generate test cases and score your performance, comparing to the optimal solution when plausible to compute.

Introduction

In this project, you will explore a known NP-complete problem. The sections below detail each of the project components. This is a group-level project with group sizes of 2 to 4 students.

Part A

Group and Project Selection

Form a group of 2 to 4 students (3 is the target value). For groups of 2, you will not have a *reduction solution architect*. For groups of 4, you will be required to have two *approximation solution architects*. Your group will select and rank three NP-complete problems that you would like to work on for this project from the list below. You should do some research before selecting one of these problems.

The problems you can explore are the following:

- Max 3-Sat
- Max clique
- Clique cover
- Traveling Salesman Problem (TSP) with a complete graph
- Longest Path

- Min Vertex Cover
- Min Graph Coloring
- Facility Location Problem
- Bin packing
- Max Cut

I have created example input and output files for each problem (available via links on Canvas). After I receive and review all the group preferences, I will assign each group a single NP-complete problem to work on.

Roles

Here are the roles available in the project. Each role will present their work in a class presentation.

Exact Solution Architect

Develop code that provides the optimal solution to the problem you are studying. Develop test cases of varying sizes and illustrate how the runtime (wall clock) of your program changes. This role will also outline the problem itself during the presentation.

Approximation Architect

Develop Python code that provides an approximation of the optimal solution to your problem. The code **must** employ an *anytime* approach and include a stochastic/random component.

Reduction and Bounds determination

Develop code that accepts an input instance of the problem and then determines an upper bound (maximization problem) or a lower bound (minimization problem).

The second part of this role is to write code that will reduce the input of your problem to an instance of another group's NP-Hard problem.

Exact Solution Architect Details

Coding

Develop a Python program that computes the optimal solution to the problem (in other words, its an algorithm). This program is expected to take ***exponential time*** (maybe even more time). The input and output format for this problem is specified in Canvas.

You will need to create many (50+) test cases and benchmark your cases with respect to wall clock time and the actual solution. You must show a plot of your results highlighting how input size is related to wall clock time. Be thoughtful of input size, for example, for a graph, how does changing the number of edges (or the ratio of vertices to edges) impact the runtime.

Use of AI

You are encouraged to use AI to help generate a driver program (to run all the test cases on your program) and to generate the test cases themselves.

Presentation

You will be required to present the following information about your problem:

- Describe the problem and the practical applications of the problem.
- Details on the inputs/outputs (pictures are a good idea).
- Example of a slight modification to the problem or a restriction on the problem that makes it possible to compute the answer in polynomial time.
- The reduction from a known NP-hard problem to the problem you are using in your project (i.e., Known-NP-Hard-Problem $\leq_p$ Your Problem). Show how the reduction works (again pictures) and also convince us that it runs in polytime and is correct (recall that it is an iff condition, so, you need to show both directions).
- Approach for solving your program (brute force, but how) and show/justify its analytical runtime.
- Analyze how changing the input order may have made the problem run faster. For example, consider the rocket construction problem. Trying the larger pieces first and working with the remaining rocket sections makes the program backtrack sooner than later and greatly impacts the runtime).
- Plots showing the runtime of your solver on varying size inputs. Some inputs should require your solver to run for at least 60 minutes.

Submission

Create a folder named **exact_solution** with the GIT repository. Place the following within this folder:

- Create a README.txt file that contains the following information:
  - The $\mathcal{O}$ running time. Be as specific as possible. For example, if you are studying a graph problem, express the bounds in terms of V and E (and maybe even a ratio of these values). If your problem is defined by a number of variables and a number of clauses, be specific (m is the number of clauses and n is the number of variables).
  - An example of calling your program that works with a sample input file. Cite the complete command line.
- Your code
- Your presentation in PDF format.
- Create a test_cases subfolder that contains all test cases. Create a shell script named *run_test_cases.sh* (placing it in this folder) that executes your program on all test cases. An example set of shell scripts for performing this work is provided (thanks to Martin Nester for developing these very nice scripts).

Extra Credit

For 10 extra points (out of 100) you can parallelize your code so that it uses multiple processes (easier in Python) to compute the optimal solution. Provide a command line argument (*-p*) for specifying the degree of parallelism. Your plots/analysis should include trying several values for the degree of parallelism and show how your solutions scales as you add more processes.

Rubric

Item	Points
Solver	25
Solver write up	10
Test cases	25
Presentation	40
Parallelism	10

Approximation Solution Architect

This section outlines what is expected from the approximation solution. If your group has more than 3 people in it, two people are expected to do this part

independently (that is, develop independent approximations). Sharing of test cases and other code is allowed.

Coding

Develop a Python program(s) that provides an approximation of the optimal solution to the problem. This program must run in *polynomial time*. The input and output format for this problem is specified in Canvas and is the same as the format for the exact/optimal solution code.

You must take 25 test cases from the *optimal solution architect* and show the perform (runtime and quality of solution) on these cases. Some of these cases must show that your approximation did not compute the optimal answer. If you can not find test cases that do this, you must create additional test cases.

You then must develop an additional 50 test cases that showcase problems where the optimal solution has no chance of finishing (because the runtime would be too long).

Anytime requirement

An *anytime* procedure runs a process that produces varying results as many times as possible, keeping the highest quality solution encountered. The procedure terminates at either the conclusion of some time limit/deadline or when the process receives a message/signal. Your code must accept a command line argument *-t* that tells your program how many seconds it has to execute. Your code **must** terminate in under this amount of time with an answer.

Greedy

If you propose a greedy strategy, it must break ties **randomly**, which will provide some variance in the quality of the solution.

Presentation

You will be required to present the following information about your problem to the class:

- Describe the approach to the approximation. Discuss strategies used (greedy, random, etc.).
- Analytical runtime $\mathcal{O}$ for constructing a single solution. Don't worry about the number of times your code is run because of the *anytime* parameter.
- plots that compare the result/solution of your exact solution versus the approximation on your test cases. This is where you show the value achieved

by both and see how well your approximation did versus the optimal solution. You can show this as a function of runtime of your program too (the longer your program runs, how does the quality of your solution evolve).

Submission

Create a folder named **approx_solution** with the GIT repository. If your group has two people performing this role, create two folders: **approx_solution_1** and **approx_solution_2**. Place the following within this folder:

- Create a README.txt file that contains the following information:
  - The $\mathcal{O}$ running time. Be as specific as possible. For example, if you are studying a graph problem, express the bounds in terms of V and E (and maybe even a ratio of these values). If your problem is defined by a number of variables and a number of clauses, be specific (m is the number of clauses and n is the number of variables).
  - An example of calling your program that works with a sample input file. Cite the complete command line.
- Your code
- Your presentation in PDF format.
- Create a test_cases subfolder that contains all test cases. Create a shell script named *run_test_cases.sh* (placing it in this folder) that executes your program on all test cases. Clearly label (with comments) which test case causes your program to run for more than 60 minutes. An example set of shell scripts for performing this work is provided (thanks to Martin Nester for developing these very nice scripts).

Extra Credit

For 10 extra points (out of 100) you can parallelize your code so that it uses multiple processes (easier in Python) to increase the number of candidate answers you can generate in the given time window. Provide a command line argument (*-p*) for specifying the degree of parallelism. Generate plots that show several values for the degree of parallelism and show how your solutions scales as you add more processes.

Rubric for Approximation

Item	Points
Solver	30
Solver write up	10
Test cases	25
Presentation and Plots	35
Parallelism	10

Reduction Solution Architect

Coding

Develop a Python program that transforms/reduces your program into another known NP-Hard problem that another group is working on in the class. If you feel this is too difficult, please consult me and we can agree on a different problem to reduce to.

This program is expected to run in *polynomial time* . The input and output format for the problem to which you are reducing is specified in Canvas.

Code Bounds for Input Problems

Develop code to compute an upper or lower bound (as appropriate). This code must run in *polynomial time*.

Use of AI

You are expected to create at least 50 test cases to verify the correctness of your code and also the runtime characteristics. You can work with the other members of your group and it is encouraged that you use these test cases from both the optimal and approximation cases. You are encouraged to use AI to help generate a driver program (to run all the test cases) and the test cases themselves. You must show a plot of your results highlighting how input size is related to wall clock time. Be thoughtful of input size, for example, for a graph, how does changing the number of edges (or the ratio of vertices to edges) impact the runtime.

Presentation

You will be required to present the following information about your problem to the class:

- Describe the reduction, clearly showing how the pieces of your problem are reduced.

- Show the details on why this solves your initial problem.
- Approach for establishing your bound
- Plots that show the optimal solution test cases that show the optimal value and your bound for varying size problems.
- Plots that show the approximation found and how that compares to your bounds.

Create a *README* file to go with your code. This file contains:

- The $\mathcal{O}$ running time. Be as specific as possible. For example, if you are studying a graph problem, express the bounds in terms of V and E (and maybe even connectiveness). If your problem takes a number of variables and a number of clauses, be specific (m is the number of clauses and n is the number of variables).
- An example of calling your program that works with a sample input file. Cite the complete command line.

Submission

Create a folder named **reduced_solution** with the GIT repository. Place the following within this folder:

- Create a README.txt file that contains the following information:
  - The $\mathcal{O}$ running time. Be as specific as possible. For example, if you are studying a graph problem, express the bounds in terms of V and E (and maybe even a ratio of these values). If your problem is defined by a number of variables and a number of clauses, be specific (m is the number of clauses and n is the number of variables).
  - An example of calling your program that works with a sample input file. Cite the complete command line.
- Your code
- Your presentation in PDF format.
- Create a test_cases subfolder that contains all test cases. Create a shell script named *run_test_cases.sh* (placing it in this folder) that executes your program on all test cases. Clearly label (with comments) which test case causes your program to run for more than 60 minutes. An example set of shell scripts for performing this work is provided (thanks to Martin Nester for developing these very nice scripts).

Extra Credit

- for 5 extra credit points, you can show on a single plot the optimal, approximation, and the bounds values for 30 different problems of varying sizes.
- for 5 extra credit points, you can run your reduces problem input into another group's NP-complete code and produce plots that show the runtime and value(answer) obtained. You may enlist another group member to help you with this part.

Rubric

Item	Points
Solver	25
Solver write up	10
Test cases	25
Presentation	40
Advanced Plots	5
Reduced problem run and results	10