

Homework 2: Linear and Integer Programming

CMU 15-780: Graduate AI (Spring 2023)

Out: Feb 15, 2023

Due: Mar 1, 2023 11:59 pm

Instructions

Collaboration Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. No downloading or copying of code or other answers is allowed. If you use a string of at least 5 words from some source, you must cite the source. Do not use any third-party libraries except those explicitly stated in this handout.

Submission

You should submit two files via Gradescope: a completed `problems.py` for the programming portion, and a PDF for your answers to the written component. Your completed functions will be autograded by running through several test cases (not just the ones supplied with starter code).

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

1 Written: LP and IP [50 pts]

Give proofs or counterexamples for all claims.

1. [10 points] Integer programming is NP-hard. Find the issue with the following incorrect proof that integer programs can be solved in polynomial time. Let

$$\max \{c^T x : Ax \leq b, x \in \{0, 1\}^n\}$$

be any integer program. Let P be the convex hull of all $x \in \{0, 1\}^n$ that satisfy $Ax \leq b$. P is a polytope, so there exists A' and b' such that

$$P = \{x \in \mathbb{R}^n : A'x \leq b'\}.$$

Now, we can find the optimal solution x^* to the linear program

$$\max \{c^T x : A'x \leq b', x \in \mathbb{R}^n\}$$

in polynomial time. But since the vertices of P are integral, x^* itself must be integral, that is, $x^* \in \{0, 1\}^n$. So x^* is an optimal solution to the original integer program as well, which shows that integer programming is in P.

Simplex

The choice of variables to enter/exit the basis at any stage of the simplex algorithm can impact the number of steps until simplex finds the optimal solution. We will illustrate this by hand-executing simplex on the

following LP:

$$\begin{aligned} \text{maximize } z &= x_1 + 4x_2 + 5x_3 \\ \text{subject to } x_1 + 2x_2 + 3x_3 &\leq 2 \\ 3x_1 + x_2 + 2x_3 &\leq 2 \\ 2x_1 + 3x_2 + x_3 &\leq 4 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

You may wish to perform simplex operations using the *simplex tableau*, which is a convenient way of keeping track of the current basic feasible solution (but this is not required for the following questions). A reference on how to perform basis operations using simplex tableaus can be found at <https://math.mit.edu/~goemans/18310S15/lpnotes310.pdf>

2. [2 point] First, write the LP in standard form by introducing slack variables. Recall that an LP is in standard form if it can be written as: maximize $c^T x$ subject to $Ax = b$, $x \geq 0$.

We will start the simplex algorithm with the basis consisting of the slack variables. So x_1, x_2, x_3 are non-basic.

3. [3 point] What is the basic feasible solution corresponding to this basis? Verify that it is feasible, and compute its objective value. Explain why this solution is not optimal.
4. [5 points] We would now like to bring a new variable into the basis with the goal of improving the objective value. Bring x_3 into the basis. What variable must leave the basis to “make room” for x_3 ? Write down the resulting basic feasible solution, and compute the new objective value. Is the basic feasible solution optimal? Why or why not?

If your basic feasible solution from the previous part is not optimal, the simplex algorithm needs to perform additional pivot operations. Do not continue the execution of simplex if this the case. We will instead see what happens if we had performed a different pivot operation at the first step.

5. [5 points] Start the simplex algorithm with the slack variables as the basic variables and x_1, x_2, x_3 as the non-basic variables as before. Now, we would like x_2 to enter the basis. Which variable must exit the basis for this to be possible? Write down the new basic feasible solution, compute the new objective value, and determine (with proof) whether the solution is optimal or not.

Note: your proofs (or disproofs) of optimality should be based on primal data coming from the current iteration of the simplex algorithm (and not based on, for example, duality).

Minimax theorem for two-player zero-sum games. A two-player zero-sum game is defined by a real-valued $m \times n$ payoff matrix

$$A = (a_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}.$$

The row player picks a row of A and the column player picks a column of A (these are the *actions* available to each player). If the row player picks row i and the column player picks column j , the row player’s payoff is a_{ij} and the column player’s payoff is $-a_{ij}$. Furthermore, the players are not limited to deterministic actions. A *mixed strategy* for the row player is a distribution over rows (and similarly for the column player). If the row player plays mixed strategy x and the column player plays mixed strategy y , the row player’s payoff is

$$\sum_{i=1}^m \sum_{j=1}^n x_i y_j a_{ij} = x^T A y.$$

The *maximin* payoff \underline{v} for the row player is the highest payoff she can guarantee if she has to commit to a mixed strategy first, and the column player can consequently best respond:

$$\underline{v} = \max_x \left(\min_y x^T A y \right).$$

The *minimax* payoff \bar{v} for the row player is the highest payoff she can guarantee if the column player commits to a mixed strategy first:

$$\bar{v} = \min_y \left(\max_x x^T A y \right).$$

6. [4 points] Argue that $\underline{v} = \max_x \min_{1 \leq j \leq n} x^T A e_j$ where e_j is the vector with a 1 in the j th entry and 0 everywhere else. That is, the column player need only stick to deterministic strategies if the row player moves first.
7. [5 points] Write a linear program that captures the row player's problem of picking the mixed strategy x to commit to if she is forced to move first (so the optimal solution to your linear program should have objective value equal to \underline{v}).
8. [6 points] Use linear-programming duality to prove that $\underline{v} = \bar{v}$. This shows, somewhat counterintuitively, that it does not matter whether you decide on your strategy first or second in a zero-sum game!

BCP revisited. Recall the boolean satisfiability problem (SAT) from homework 1, which asks whether a boolean formula in conjunctive normal form (CNF) is satisfiable.

9. [4 points] Formulate SAT as an integer program. Specifically, precisely define the variables, constraints, and objective of your integer program. Prove that if there is a satisfying assignment to the SAT formula then there is a solution to your integer program, and vice versa.
10. [6 points] Let's compare running your LP relaxation to running BCP given some formula and some partial assignment midway through a search. Prove or disprove:
 - (a) If variable x_i is assigned a value by BCP, then x_i has that value in every feasible solution of the LP relaxation.
 - (b) If variable x_i has the same value in every feasible solution of the LP relaxation, then x_i will be assigned that value by BCP.

2 Programming: Combinatorial Auctions [50 pts]

You will be implementing parts of a winner determination algorithm for combinatorial auctions. Recall that a combinatorial auction consists of m items and n bids. Each bid is for a particular subset of the items. Formally, each bid is of the form (B, p) , where $B \subseteq \{1, \dots, m\}$, and p is the price of the bid. The winner determination problem is the problem of finding a set of non-overlapping bids of maximum total price.

In Python, we will represent the m items of the auction by their indices: $[0, 1, \dots, m-1]$. Each bid is represented as a tuple containing (1) a list of items and (2) a price. For example, $([0, 4], 10)$ represents a bid for the bundle of items $\{0, 4\}$ at a price of \$10. An instance of a combinatorial auction is represented by a tuple containing (1) the number of items and (2) a list of bids. For example, $(5, [([0, 4], 10), ([1], 4)])$ is an auction with five items and two bids; the first bid is for items 0 and 4 at a price of \$10 and the second bid is for item 1 at a price of \$4. Empty bids (bids with no items) are not allowed. Each bid can contain at most one copy of any item. We will assume that no two bids will be for the exact same subset of items (since it suffices to keep only the highest-price bid for any subset). All items appear in at least one bid each. Bid prices are positive integers.

You will solve this problem by formulating it as a binary integer program (BIP) and writing a generic solver for BIPs based on A^* search (branch-and-bound).

1. [10 points] Implement the function `to_integer_program(m, bids)` in `problems.py`. This function takes as input the number of items m and the list of bids `bids`, of length n , defining an auction. Your function should output a tuple of three numpy arrays (c, A, b) representing an integer programming formulation of the Branch-on-Bid method for winner determination. In particular, (c, A, b) should be such that solving winner determination is equivalent to solving the IP

$$\max \{c^T x : Ax \leq b, x \in \{0, 1\}^n\}. \quad (1)$$

Your formulation should have one variable for each bid, and the optimal value of the integer program should be the optimal revenue. Your formulation will be tested by comparing the optimal value of your program to the true optimal value on test instances.

To test this section of your code, you will need to install `cvxpy` and `cvxopt`, which you can do via `pip`. You will not be implementing A^* , nor the simplex algorithm—we have already done both of those things for you, as functions `a_star` and `simplex` respectively. You will use the integer programming formulation of the winner determination problem to implement a search problem whose optimal solution corresponds to the optimal solution of the integer program. Use any branching rule you want.

2. [20 points] **Basic integer programming solver.** Implement the functions in the `IPSearchProblem` class, so that calling `a_star(IPSearchProblem(c, A, b))` solves an integer program of the form (1). Your solver will be tested by comparing the optimal value returned by your solver to the true optimal value on test instances.

A node in the search tree is represented by a partial solution $\tilde{x} \in \{0, 1\}^S$ to the BIP, where $S \subseteq [n]$. In the language of A^* , the $f(\cdot)$ -value should be the value of the LP relaxation of the heuristic:

Some considerations to be aware of:

- (a) Beware of floating-point arithmetic. Never test for equality of floats: tests for $x = y$ should be written as $|x - y| < \text{TOL}$; tests for $x < y$ should be written as $x < y - \text{TOL}$; and tests for $x \leq y$ should be written as $x < y + \text{TOL}$. The tolerance `TOL` is given in the starter code as 10^{-8} .
- (b) The given A^* implementation solves *maximization* problems. That is, f should be *decreasing* along paths. This is convenient for our problem formulation but may be backwards from what you may be used to.
- (c) The given simplex implementation solves linear programs in *standard form*:

$$\max \{c^T x : Ax = b, x \geq 0\}.$$

You should convert your linear programs to this form before supplying them to the solver.

- (d) You may choose what representation you use for nodes. The `a_star` function will work with any representation.
 - (e) The simplex LP solver will return $-\infty$ as the value if the LP is infeasible, and $+\infty$ if the LP is unbounded. The latter case should not happen; you should handle the former case in your code.
 - (f) You can assume that c, A, b will always be integers.
3. [20+ points] **Faster integer programming.** Speed up your integer programming solver. This problem is very open-ended, and you can take it in several different directions. You should write your solver entirely using pure Python and `numpy`—do *not*, for example, call `cvxpy`'s linear program solvers. You will be graded on how many of our instances your solver is able to solve in a fixed amount of time.

Full credit will be given to solutions that solve 150 instances in the time limit given, and extra credit is available. Let s be the number of test instances your solver completes in the given time limit. Your score on this problem will be given by $\min(40, \max(0, s/2 - 55))$ where s is the number of problems you solve in the time limit. Our reference implementation scores $s = 150$, and it does not use all the below optimizations.

You should submit only your fastest solver in `problems.py`—assuming it is correct, it will be given the 20 points for the previous part.

Warning: If you're benchmarking your own code, make sure you're testing your `IPSearchProblem` implementation, not `cvxpy`'s MIP solver!

Some ideas for speeding up the solver include the following, in (very) rough order of ease of implementation. You do not need to implement these in particular; you are free to implement other optimizations as you see fit.

- (a) Use a better heuristic to pick your branching variable. For example, you could branch on the variable whose value is closest to $1/2$. Or use another heuristic—either one you devise yourself, or one from the lecture notes!

- (b) Switch to using simplex on the *dual* of your LPs, and use the returned LP basis to warm-start subsequent LP solutions. This should massively speed up your solver already.

When using simplex on the dual, the solution returned by `simplex` will be, of course, a dual optimal solution, not a primal optimal solution. To convert it to a primal optimal solution, you must use *complementary slackness*: given a primal-dual pair

$$\begin{array}{ll} \max \{c^T x : Ax \leq b\} & \min \{b^T y : A^T y = c, y \geq 0\} \\ \text{(primal)} & \text{(dual)} \end{array}$$

with $A \in \mathbb{R}^{m \times n}$, a dual-optimal basis $I \subseteq [m]$, the primal-optimal solution is given by $x^* = A_{I,:}^{-1} b_I$, where $A_{I,:}$ denotes A restricted to the rows with indices in I . Do *not* explicitly invert the matrix $A_{I,:}$ —matrix inversion is slow. Instead, use the returned matrix inverse `A_I_inv` (see the starter code for documentation).

To warm-start simplex, pass a feasible basis and corresponding matrix inverse (`I`, `A_I_inv`) to the `simplex` function. (Warning: The solver performs a simple feasibility check on the supplied basis, but the check will not catch all possible incorrect bases. If you accidentally pass an incorrect basis, the solver may have undefined behavior.)

- (c) Implement *Gomory cuts*, according to the lecture slides, or by following an external reference of your choice (if you use an external reference, please cite it in a comment in your code). This may require modification of the `simplex` algorithm.

Some other optimizations that may have a significant practical effect include, in no particular order:

- (d) Ensure that you are not needlessly copying large objects, which may be expensive.
- (e) Avoid solving the LP at a given node multiple times—that is simply wasted time.
- (f) Use `numpy` array functions whenever possible: avoid `for` loops over large lists, because in Python these are slow. Our solution code for `IPSearchProblem` contains no loops over more than two things.