LECTURE 6

Ishaan Lal

October 12, 2024

1 Introduction

Last lecture, we introduced Linear Programming (LP), Integer Linear Programming (ILP) and Mixed-Integer Programming (MIP). Today, we will dive deeper into MIP.

Recall:

- **LP** involved maximizing/minimizing linear objective functions subject to linear inequalities
- **MIP** is identical to LP, with the extra constraint that some variables must be integers.

LP is poly-time solvable. MIP is NP-Complete.

1.1 Disclaimer

Apologies in advance for the density of these notes. This week was especially difficult and I had to veer from my normal method of creating notes.

1.2 Pointwise Discontinuity

We seem to have a good grasp of how to solve LP-related problems when the objective function is indeed Linear. But we may choose to ask: *What do we do when the objective is discontinuous and linear?* or perhaps *What do we do when the objective is piecewise linear?* These are both valid questions, and may arise in many real world applications.

Consider the following setting:

Problem Setting

You are the proud owner of your business called *Quackulus*, where you specialize in creating novel rubber ducks. Suppose it costs \$10 to produce a single duck. There is also a fixed setup cost of \$250 if you choose to produce any units. Additionally, you can only create a maximum of 1000 ducks.

You are aiming to minimize your cost of production subject to some unknown linear constraints.

Given the setting, naturally, our objective function would be formalized as:

$$\text{minimize} \quad 250 + 10n$$

where $n \in \mathbb{N}$ represents the number of ducks we produce. However, when $n = 0$, our objective function evaluates to 250, when it should be 0. Thus, our correct function is:

$$\begin{cases} 0, & n = 0 \\ 250 + 10n & n > 0 \end{cases}$$

This is **NOT** a MIP because our objective function is not linear in the domain. There is a discontinuity at $n = 0$. MIP requires our objective function to be linear. How do we fix this?

- **Idea 1:** Add a constraint of $n > 0$. This doesn't work, because we definitely would want to consider the option of producing $n = 0$ ducks.
- **Idea 2:** Still add the constraint of $n > 0$, then solve the MIP, which would indicate the "best solution given that we produce a duck", and later compare that with producing 0 ducks, and choose the better option.

  This can work, but think about what happens if we have a more complex discontinuous linear function. Things could become impractical. Also consider what would happen if the discontinuity was somewhere in the middle of the domain (instead of at one of the bounds).

Solution

Notice that the number of ducks we can produce is at most 1000. So if we choose to produce ducks, then $n \leq 1000$, otherwise, $n = 0$. To formalize this, we will introduce an indicator variable z whereby:

$$z = \begin{cases} 1 & \text{we make ducks} \\ 0 & \text{we do not make ducks} \end{cases}$$

We can now reformulate our objective as a linear function:

$$\begin{aligned} \text{minimize} \quad & 250 \cdot z + 10 \cdot n \\ \text{subject to} \quad & n \leq 1000 \cdot z \\ & n \geq 0 \\ & z \in \{0, 1\} \\ & \text{other constraints} \end{aligned}$$

1.3 Piecewise Linear Functions

We now consider a different scenario, where our objective function is best described as a piecewise function.

Problem Setting

Quackulus has undergone some improvements where the cost of production has changed. Now, there is no fixed set-up cost. However, the cost per unit depends on the number of units produced.

The first 400 ducks you produce will cost \$5 each to produce. The next 200 ducks will cost only \$2 each. And the next 400 ducks will cost only \$3 each.

For example, if you choose to create 500 ducks, it will cost you:

$$400 \cdot \$5 + 100 \cdot \$2 = \$2200$$

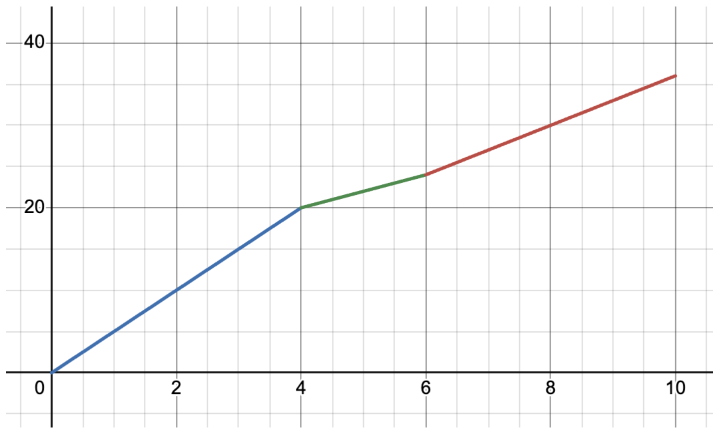
And if you choose to create 900 ducks, it will cost you:

$$400 \cdot \$5 + 200 \cdot \$2 + 300 \cdot \$3 = \$3300$$

Under this setting, we can write the objective function as a piecewise linear function:

$$\begin{cases} 5 \cdot n & 0 \leq n \leq 400 \\ 5 \cdot 400 + 2 \cdot (n - 400) & 401 \leq n \leq 600 \\ 5 \cdot 400 + 2 \cdot 200 + 3 \cdot (n - 600) & 601 \leq n \leq 1000 \end{cases} = \begin{cases} 5n & 0 \leq n \leq 400 \\ 2n + 1200 & 401 \leq n \leq 600 \\ 3n + 600 & 601 \leq n \leq 1000 \end{cases}$$

Graphically, our objective function looks as follows, where the x-axis represents the number of hundreds of ducks created, and the y-axis represents the cost in hundreds of dollars:



Similar to how we handled discontinuity, we'll leverage the property of indicators. Here, indicators would be helpful to indicate which segment of the curve we are on.

Consider variables $\delta_1, \delta_2, \delta_3$ where $n = \delta_1 + \delta_2 + \delta_3$ and where $0 \leq \delta_1 \leq 400$, $0 \leq \delta_2 \leq 200$, $0 \leq \delta_3 \leq 400$.

δ_i represents how far into the ith piece (segment) of the piecewise function we are. For example if $n = 500$ ducks, then $\delta_1 = 400$, $\delta_2 = 100$, $\delta_3 = 0$, indicating that we have passed the first segment (since δ_1 is at its max) and are 100 units in to the second segment. With this formulation, it is easy to see that:

$$\text{Cost} = \$5 \cdot \delta_1 + \$2 \cdot \delta_2 + \$3 \cdot \delta_3$$

Our objective function is now linear, but we must codify this idea of the definition of δ_i into our constraints. We do this as follows: Note that δ_2 is contingent on δ_1, as $\delta_2 > 0$ only if $\delta_1 = 400$ (at its max). Similar for δ_3 being contingent on δ_2.

Let i_1 be an indicator that is 1 if δ_1 has reached its upper limit. Let i_2 be an indicator that is 1 if δ_2 has reached its upper limit.

Then, if $i_1 = 0$ we must have $0 \leq \delta_1 \leq 400$, but if $i_1 = 1$, it must be that $\delta_1 = 400$, or equivalently, $400 \leq \delta_1 \leq 400$. We can combine this into a single constraint as follows:

$$i_1 \cdot 400 \leq \delta_1 \leq 400$$

Similarly, if $i_2 = 0$, then $0 \leq \delta_2 \leq 200$, and if $i_2 = 1$, then $200 \leq \delta_2 \leq 200$. As a single constraint:

$$i_2 \cdot 200 \leq \delta_2 \leq 200$$

But recall that $i_2 = 1$ only if $i_1 = 1$. How do we incorporate this? One small change is needed:

$$i_2 \cdot 200 \leq \delta_2 \leq 200 \cdot i_1$$

As an exercise, take some time to convince yourself why this constraint enforces the implication we had.

Following the pattern from above, we can only have $\delta_3 \geq 0$ if $i_2 = 1$. Then, our last constraint is:

$$0 \leq \delta_3 \leq 400 \cdot i_2$$

Our fully formalized MIP is:

$$\begin{aligned} \text{minimize} \quad & 5\delta_1 + 2\delta_2 + 3\delta_3 \\ \text{subject to} \quad & i_1 \cdot 400 \leq \delta_1 \leq 400 \\ & i_2 \cdot 200 \leq \delta_2 \leq i_1 \cdot 200 \\ & 0 \leq \delta_3 \leq 400 \cdot i_2 \\ & i_1, i_2 \in \{0, 1\} \end{aligned}$$

1.4 Towards Continuity

But of course, the most fascinating question is what happens when our objective function is continuous? While we will not go very in-depth into this question, a simple idea is as follows: We can approach the optimal solution by approximating the continuous curve of the objective function as a piecewise linear function (with many, but finite, number of segments). From here, use the approach outlined in the previous section to get an approximation of the solution.

2 How MIP Solvers Work

We'll now take some time to try to develop an algorithm to solve MIP. Remember, MIP is an NP-complete problem. MIP solvers use a technique called "Branch and Bound", which we will get to shortly. First, we need to build our way up.

For simplicity, we will assume that the variables involved in our MIP have an upper and lower bound:

$$lb(x) \leq x \leq ub(x)$$

Keep in mind, in most applications of MIP, some bound would be present (for example, it is almost certain that I will create less than 10^{15} ducks).

Additionally, we will assume that we are solving a "maximization" problem. "Minimization" problems work similarly, just some things will be mirrored.

As per usual, we can treat our search space as a tree, where from one node, we traverse down a branch to a lower level when we assign a new variable. At the leaves, all variables have been assigned, and each leaf corresponds to a different assignment.

2.1 Naive Branching

Let P represent the objective function of our MIP. A first approach to solving MIP would be as follows:

- Choose a variable and split its viable domain in half
- Generate subproblems on each "half"
- Solve the subproblems recursively
- Choose whichever subproblem has the higher (or lower, if minimizing) objective value, and discard infeasible solutions.

In terms of code, our initial algorithm is given as the following. Note that “lb” represents “lower bound” and “ub” represents “upper bound”.

```
naive(P):
    if lb = ub for all vars:
        if $P$ violates a constraint:
            return INFEASIBLE
        return objective_value($P$)
    let $x$ be a variable with lb(x) < ub(x)
    let $m = \lfloor (lb(x) + ub(x))/2 \rfloor$
    return max{naive(P | $x \leq m$), naive(P | $x \geq m$)}
```

With any algorithm, we tend to be curious about its runtime. What is the runtime of the above code? It may be a bit difficult to compute. But a critical observation is that **this algorithm will only terminate for pure integer programs**.

But here is an even more pressing issue: we essentially have to explore our entire search tree. That is, in our recursion statement, we have to check both branches. This is noticeably different from our naive backtracking algorithm for DPLL, because with that algorithm, we could stop immediately when we found a solution (since it was a decision problem). Here, we explore everything. Another key difference is that DPLL had inference.

2.2 Adding Inference

To add an element of inference to our solver, we recall LP relaxation. Consider the following idea: for a MIP P, we get its **LP relaxation** $LP(P)$ by allowing all variables to be fractional. Recall from last lecture that after doing so, we cannot just round our solution to get the MIP solution – this does not always work. Instead, we make the key observation:

Key Observation

The LP solution is always at least as good (aka equally good, if not better) as the MIP solution (with respect to objective value)

Idea: Since LP is poly-time solvable, we will use an LP solver as an inference tool. Instead of recursing until all variables have one value, solve $LP(P)$ and check whether all integer variables have integer values. We can branch on integer variable x whose value v is fractional in $LP(P)$, and create subproblems of $x \leq \lfloor v \rfloor$ and $x \geq \lceil v \rceil$

While we are at it, we will also discuss *pruning the search tree.* Just as we did with DPLL, when we found a decision that would yield a conflict, we stopped searching down that subtree. We want to do something similar here.

Idea: discard partial solutions that will never yield a better objective value than one we've already found. If we've seen a MIP solution with a better objective value than LP(P), discard P since any integer solution can only be worse than LP(P).

2.3 Branch and Bound

This idea we have constructed of branching our solution space via LP and pruning is called the **“Branch and Bound”** algorithm. The first version was developed by Alisa Land and Alison Harcourt in 1960.

Refer to the slides for the pseudocode of Branch and Bound, as well as a visual example.

2.4 Choices

What are the choices we, as the algorithm designer, can make when implementing branch and bound?

- Which subproblem should we visit?
  - We could visit the oldest existing subproblem (a la BFS)
  - We could visit the most recent subproblem (a la DFS)
  - We could visit the subproblem with the best LP objective ("best-first search)
- Which variable should we branch on?
  - The most constrained variable (smallest domain)
  - The variable with the largest/smallest coefficient in the objective function
  - The variable closest/farthest to halfway between integers

Each choice has their own benefits depending on the problem. Most solvers allow users to tune the solver (make choices) based on knowledge of the problem.

2.5 Branch and Cut

Briefly, a **cut** for a MIP P is a new constraint that does not eliminate any feasible solutions for P, but does for $LP(P)$.

The Branch and Cut algorithm, proposed by Manfred Padberg and Giovanni Rinaldi in 1989, finds cuts of MIP, then adds them, and recurses on the new MIP. The motivation is that tighter LP relaxation means we converge faster to the MIP solution.

How to find cuts is beyond the scope of this class.

As a corollary: if all integer variables take integer values in the optimal solution to $LP(P)$, then it is also the optimal solution to $MIP(P)$.

The intuition behind the observation is that a solution to MIP is also a solution to LP ($MIP \subset LP$, in a way).

3 The Knapsack Problem

The **Knapsack Problem** is described as follows: Given n items with values $v_1, ..., v_n$ and weights $w_1, ..., w_n$, select the maximum-value subset of items to fit into a knapsack with capacity W.

For the sake of example, suppose we had the following items:

- A \$500 bill weighing 0.5oz
- A gold bar, worth \$4,000 weighing 300oz
- A diamond ring, worth \$5,000, weighing 1oz
- An antique pot, worth \$5,000, weighing 200oz
- A gold coin, worth \$2,000, weighing 100oz

Suppose our backpack can carry a maximum weight of 400oz, and we want the contents to be of highest value.

3.1 Fractional Knapsack

Suppose I could choose to take fractions of the items. For example, if I wanted to take $\frac{1}{3}$ of the antique pot, I could do that. Its valuation would then be $\frac{1}{3} \cdot 5000$ but it would weight only $\frac{1}{3} \cdot 200$oz.

How would we solve this problem? What do we want to prioritize? The most valuable items? The lightest items? Something else?

It turns out that this problem is solved, and has a very simple solution. The solution is a **greedy algorithm** that sorts the items by value-to-weight ratio, and takes as much of each item as possible, in order, until the knapsack is full.

With our example, the bill has ratio $500/0.5 = 1000$, the gold bar has ratio $40/3 \approx 13.3$, the diamond ring has ratio 5000, the pot has ratio 25, and the coin has ratio 20.

Following the algorithm, we would choose to include the entire ring, as it has the highest ratio, leaving us with 399oz in the bag. Then, we would choose the dollar bills, leaving us 398.5oz. Next would be the pot, leaving 198.5oz. Then the coin, leaving 98.5oz. And finally, we would choose $\frac{98.5}{300}$ of the bar.

This algorithm runs in $\mathcal{O}(n \lg n)$ time, where n is the number of items.

3.2 0/1 Knapsack

Now consider the situation where we **cannot** choose fractional items. That is, for each item, we either include the full thing, or we do not. Here, the greedy algorithm fails. The 0/1 Knapsack problem is NP-complete.

Luckily, we can solve 0/1 Knapsack by formulating it as an instance of MIP, and the formulation is quite straightforward:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} x_i v_i \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,1\} \quad \forall i \end{aligned}$$

Other solutions to 0/1 Knapsack exist. For example, there is a dynamic programming (DP) solution that runs in $\mathcal{O}(nW)$ time. However, this is not polytime, because W could be exponential with respect to n.

There is also an approximation algorithm, which guarantees a solution that is at least 50 percent of the optimal solution, and it runs in time $\mathcal{O}(n \lg n)$. While the polynomial is nice, the approximation is quite poor.

While MIP is not a poly-time algorithm, it like DP, can be useful depending on the parameters.

Further, we can use the Branch and Bound paradigm to solve Knapsack without even treating it as an instance of MIP. See the slides for the pseudocode.

4 References

1. MIT Notes