LECTURE 7

Ishaan Lal

February 28, 2025

1 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.

1.1 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 = ⌊(lb(x) + ub(x))/2⌋
    return max{naive(P | x ≤ m), naive(P | x ≥ 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.

1.2 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).

1.3 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.

1.4 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.

1.5 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).

2 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.

2.1 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.

2.2 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.