CIS 189

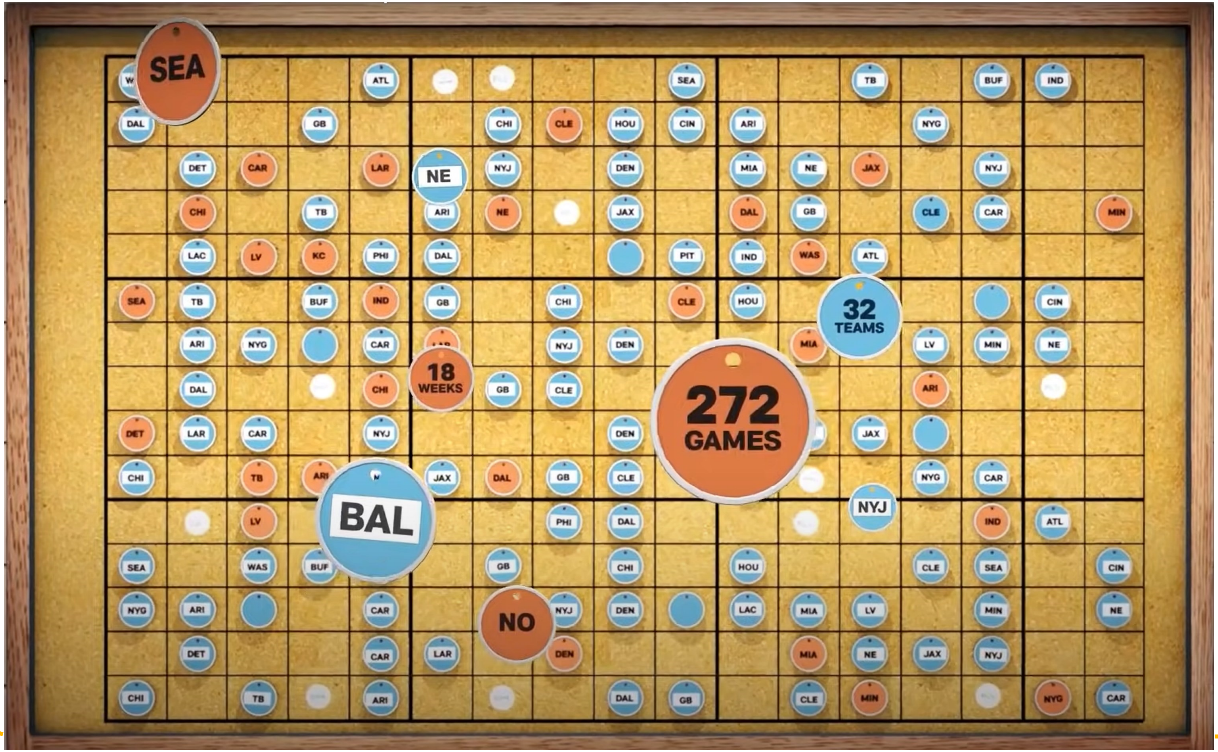
Lecture 6: More Mixed-Integer Programming

Rohan Menezes rohanmenezes@alumni.upenn.edu

Logistics

- **Homework 3:** Kidney Exchange Program
  - Due after break: 3/14 at 4pm
  - Use MIP to build a model that saves lives IRL!
- No class on 3/7 (spring break)
- Next week: guest lecture!

Next Week



Recap: LP and MIP

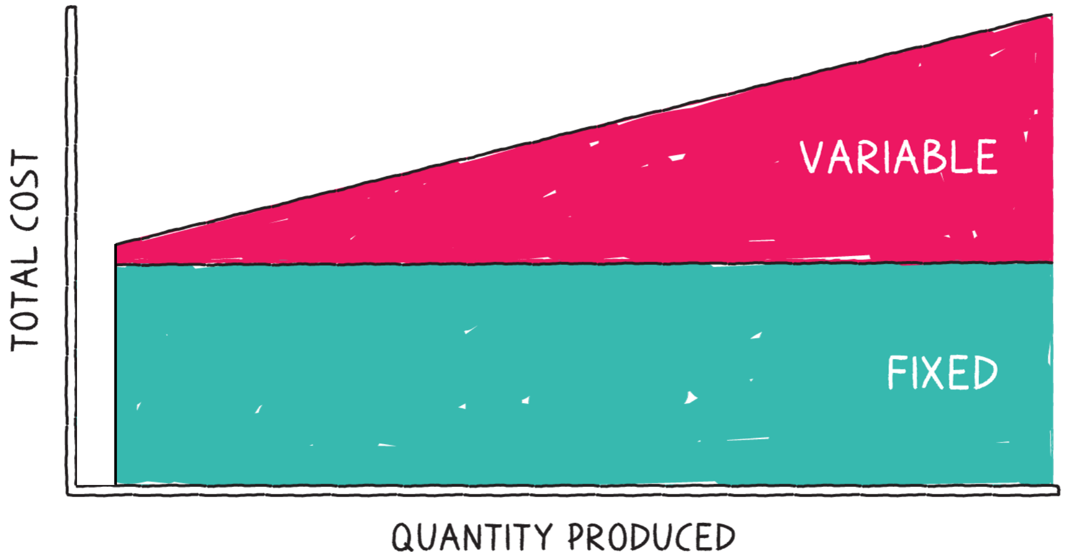
- **Linear programming:** maximize/minimize linear objective subject to linear (in)equalities
- **Mixed-integer programming:** same as linear programming, but some variables can take on integer values only
    - NP-complete!

Modeling Fixed Costs

- Suppose it costs $10 to produce each unit of a product
- Also **fixed setup cost** of $250 if we produce any units

$$\text{cost to produce } n \text{ units} = \begin{cases} 0, & n = 0 \\ 250 + 10n, & n \geq 1 \end{cases}$$

Modeling Fixed Costs



Looks nonlinear... how to model with MIP?

Indicators for Constraints

- Idea: want to create a 0/1 **indicator variable** c where $c = 0$ if we don't produce any product and $c = 1$ if we do, i.e.

$$c = \begin{cases} 0, & n = 0 \\ 1, & n \geq 1 \end{cases}$$

- Then we can model the fixed cost as:

$$\text{cost to produce } n \text{ units} = 250c + 10n$$

Indicators for Constraints

- More generally, can create indicator c for constraint $n \geq b$ if we have bounds $L \leq n - b \leq U$
    - Can replace n with any linear expression $a_1 n_1 + a_2 n_2 + \cdots + a_k n_k$, but it needs to be integer-valued
- To enforce $(c = 1) \Rightarrow (n \geq \mathrm{b})$, add constraint:

$$n - b \geq L(1 - c)$$

- To enforce $(c = 0) \Rightarrow (n \leq \mathrm{b} - 1)$, add constraint:

$$n - b \leq (U + 1)c - 1$$

Modeling Fixed Costs

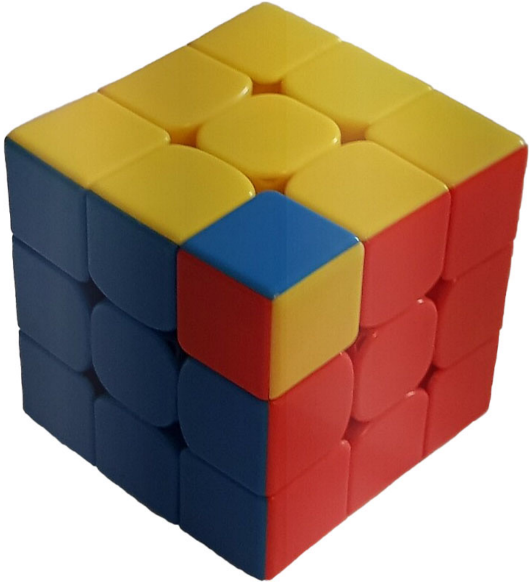
- So to make an indicator c for $n \geq 1$, add:

  $$n \leq (U+1)c$$

- If minimizing cost, don't need to enforce $(c = 1) \Rightarrow (n \geq 1)$
  - Why? Equivalent to $(n = 0) \Rightarrow (c = 0)$
  - Since cost is $250c + 10n$, solver will set $c = 0$ if possible when minimizing

Debugging Integer Programs

- Your model is `INFEASIBLE` when it shouldn't be… what to do?
- Want to find which buggy constraint(s) cannot be satisfied



Debugging Integer Programs

- Typical model has thousands, even millions of constraints
- **Insight:** bugs usually happen at the level of groups of constraints, not individual constraints

```
# Each job gets exactly one CPU
for j in range(num_jobs):
    solver.Add(
        sum(x[c, j] for c in range(num_cpus)) == 1
    )

# Each cluster gets at most two jobs
for cluster in clusters:
    solver.Add(
        sum(x[c, j] for c in cluster for j in range(num_jobs)) <= 2
    )
```

group

group

Debugging Integer Programs

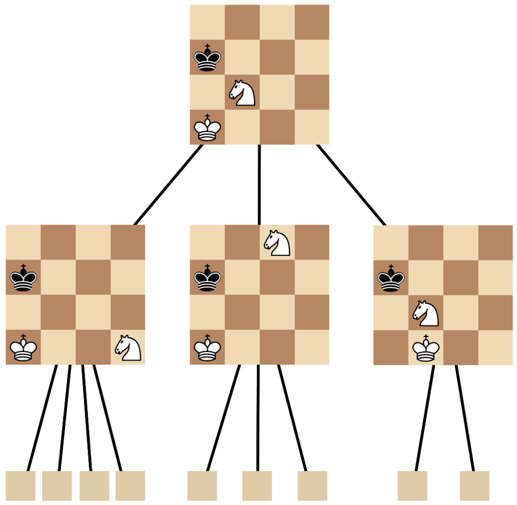
- If we get rid of all buggy constraint groups, the model should become feasible
- **Strategy:** remove groups one-by-one until model is feasible, then add them back to find minimal set of buggy groups
    - Even better: use a “binary search” strategy (remove half the constraint groups at a time)
- See demo (mip_debugging.py)

Debugging Integer Programs

- What if the model is feasible, but the solution is wrong?
- If it's easy to see that a constraint is violated, check that one
- Otherwise, just add constraints enforcing a known "right" solution, and then model will become infeasible
  - If you don't have a known solution, enforce whatever property is violated in the wrong solution (e.g. `objective <= 300`)

How do MIP solvers work?

- Most fundamental technique: **branch and bound**
  - Chess engines work using branch and bound too ("alpha-beta pruning")
- For simplicity, let's assume that all integer variables have lower and upper bounds
  - $\mathrm{lb}(x) \leq x \leq \mathrm{ub}(x)$



Naive Branching

- Want to solve MIP P where integer variables are bounded
- What's a first step for tree traversal of the search space?
- **Idea:** split the domain of a variable in half
  - Generates subproblems which can be solved recursively
- Pick whichever subproblem has the higher objective value, and discard infeasible solutions

Naive Branching (Pseudocode)

```
# find the optimal objective value for P

naive(P):
    if lb = ub for all vars:
        if P violates a constraint:
            return INFEASIBLE (-inf)
        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)}
```

How bad is Naive Branching?

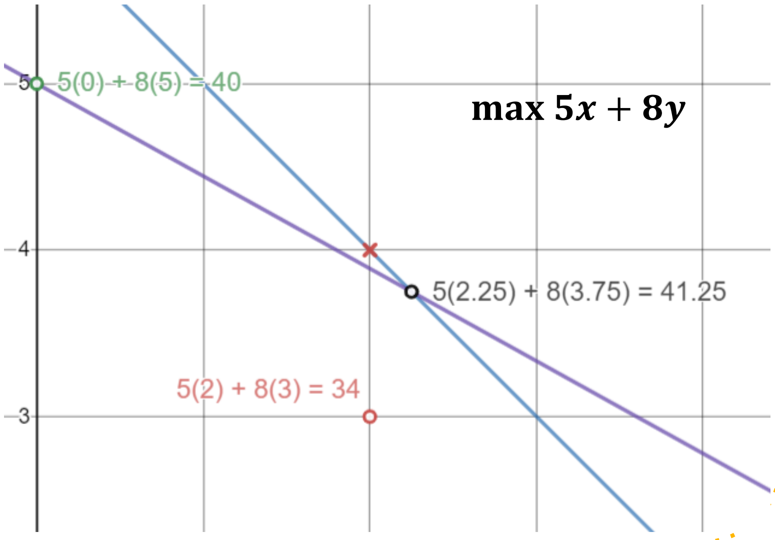
- Does naive branching even terminate?
  - Only for pure integer programs!
- Which assignments does the algorithm discard or visit?
  - Need to evaluate both branches -- visits all feasible solutions!
- Basically the same as brute force
- Runtime scales with size of search space

Recall: LP Relaxation

- For a MIP P, we get its **LP relaxation** $LP(P)$ by allowing all variables to be fractional
  - Can't just round LP solution

- **Key observation:** the LP solution is always at least as good as the MIP solution (by objective value)

- Corollary: if all integer vars take integer values in optimal solution to $LP(P)$, then it is also optimal solution to P



Adding Inference

- **Idea:** since LP is polytime-solvable, use LP solver as inference engine!
- Instead of recursing until all variables have one value, solve $LP(P)$ and check whether all integer variables have integer values
- Branch on integer variable x whose value v is fractional in $LP(P)$
  - Create subproblems $x \leq \lfloor v \rfloor$ and $x \geq \lceil v \rceil$

Pruning Fruitless Nodes

- **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

Branch & Bound

- First version developed by Ailsa Land and Alison Harcourt in 1960
- Combines branching of solution space with bounds-based pruning
- B&B is an **algorithm paradigm**: a "meta-algorithm" that can be used to design algorithms for many different optimization algorithms

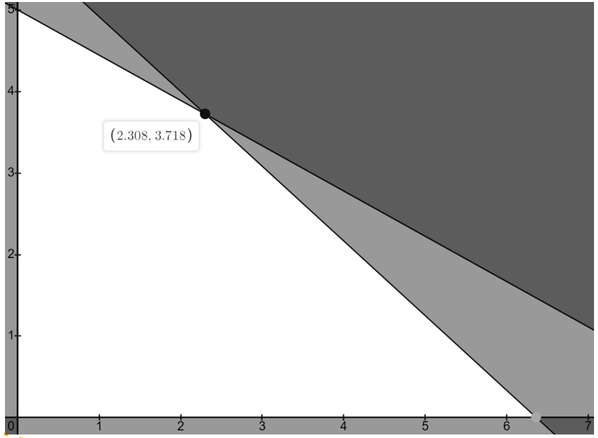
Branch & Bound (Pseudocode)

```
# find the optimal objective value for P
# best_seen is the best objective value so far
branch_and_bound(P, best_seen = -inf):
    let LP_soln = solve_LP(LP(P))
    if LP_soln = INFEASIBLE: return INFEASIBLE
    if objective_value(LP_soln) ≤ best_seen:
        return -inf
    if LP_soln satisfies integrality constraints of P:
        return objective_value(LP_soln)
    let x be an int var with fractional value v in LP_soln
    let obj1 = branch_and_bound(P|x ≤ ⌊v⌋, best_seen)
    set best_seen = max{obj1, best_seen}
    let obj2 = branch_and_bound(P|x ≥ ⌈v⌉, best_seen)
    return max{obj1, obj2}
```

Example: Branch & Bound

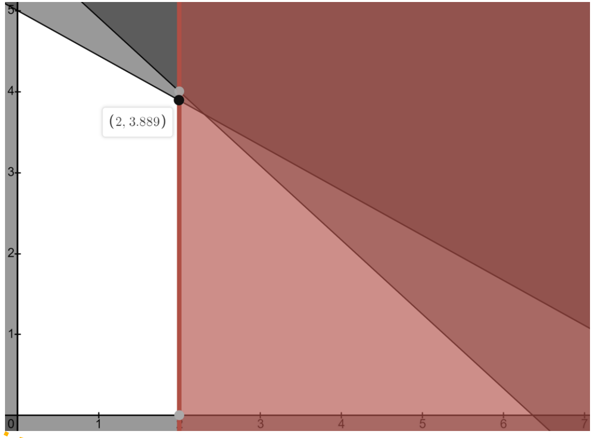
max	$f(x,y) = 5x + 8y$
s.t.	$5x + 9y \leq 45$
	$1.1x + 1.2y \leq 7$
	$x, y \in [0..100]$

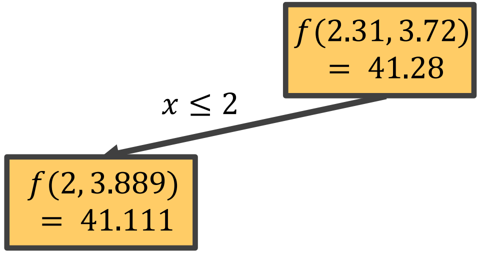
$f(2.31, 3.72) = 41.28$



Example: Branch & Bound

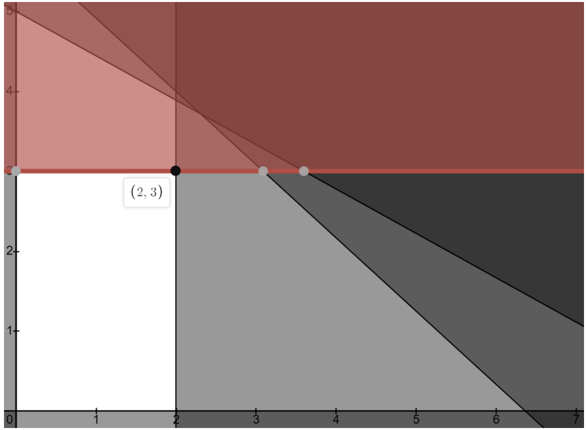
$$\begin{array}{ll} \text{max} & f(x,y) = 5x + 8y \\ \text{s.t.} & 5x + 9y \leq 45 \\ & 1.1x + 1.2y \leq 7 \\ & x, y \in [0..100] \end{array}$$

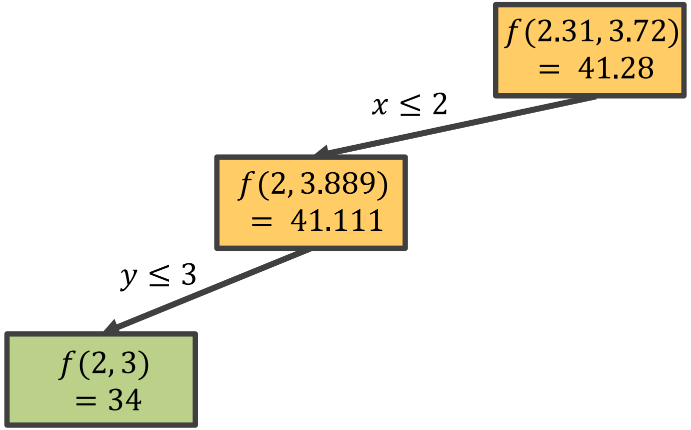




Example: Branch & Bound

$$\begin{array}{ll} \text{max} & f(x,y) = 5x + 8y \\ \text{s.t.} & 5x + 9y \leq 45 \\ & 1.1x + 1.2y \leq 7 \\ & x, y \in [0..100] \end{array}$$





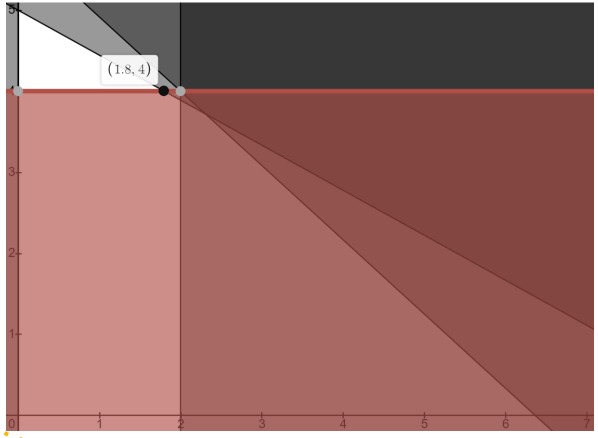
Example: Branch & Bound

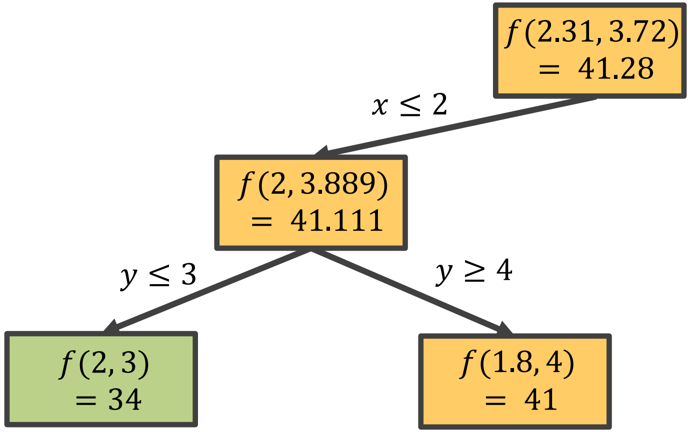
max $\quad f(x, y) = 5x + 8y$

s.t. $\quad 5x + 9y \leq 45$

$\quad 1.1x + 1.2y \leq 7$

$\quad x, y \in [0..100]$





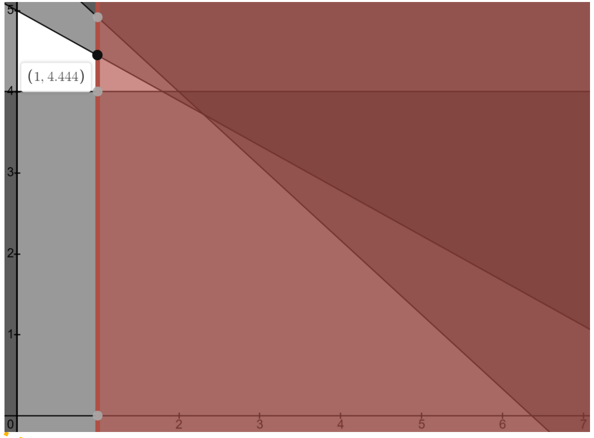
Example: Branch & Bound

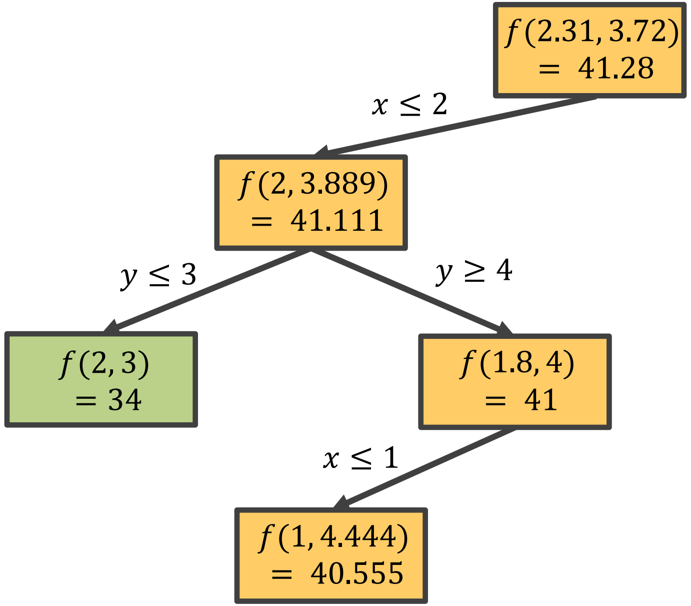
max $\quad f(x, y) = 5x + 8y$

s.t. $\quad 5x + 9y \leq 45$

$\quad 1.1x + 1.2y \leq 7$

$\quad x, y \in [0..100]$





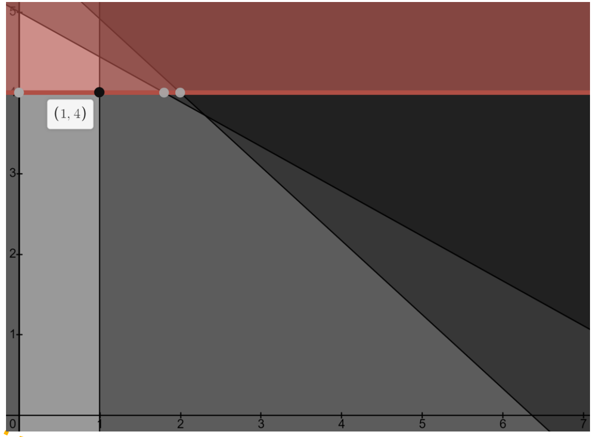
Example: Branch & Bound

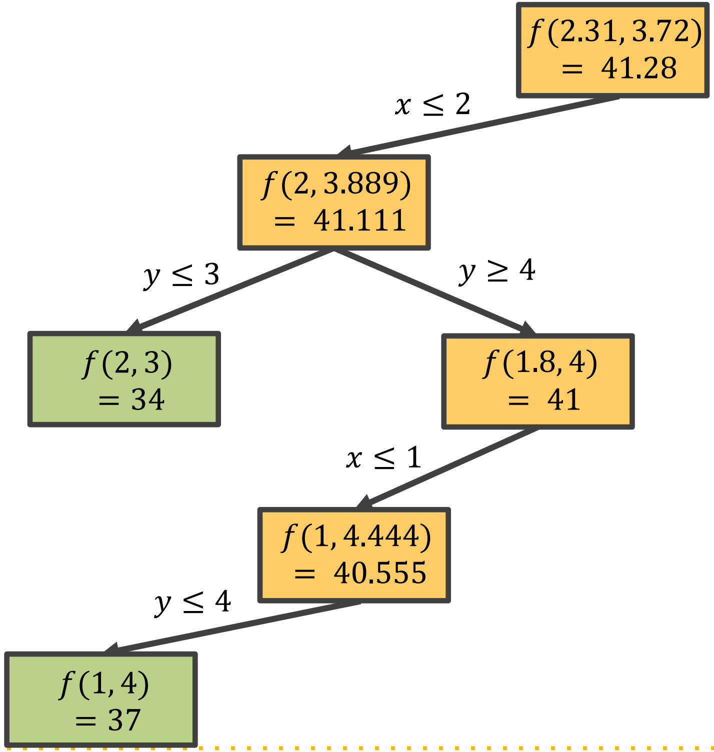
max $\quad f(x, y) = 5x + 8y$

s.t. $\quad 5x + 9y \leq 45$

$\quad 1.1x + 1.2y \leq 7$

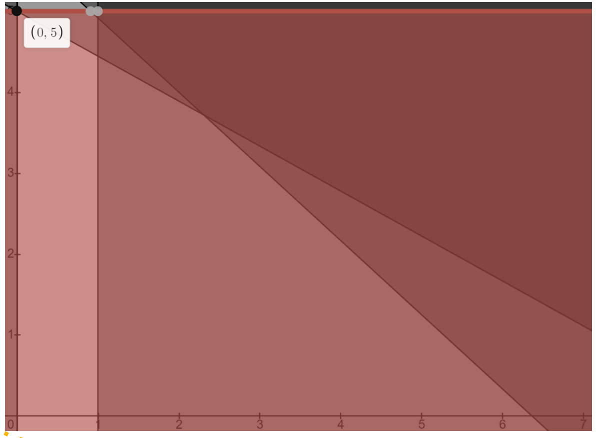
$\quad x, y \in [0..100]$

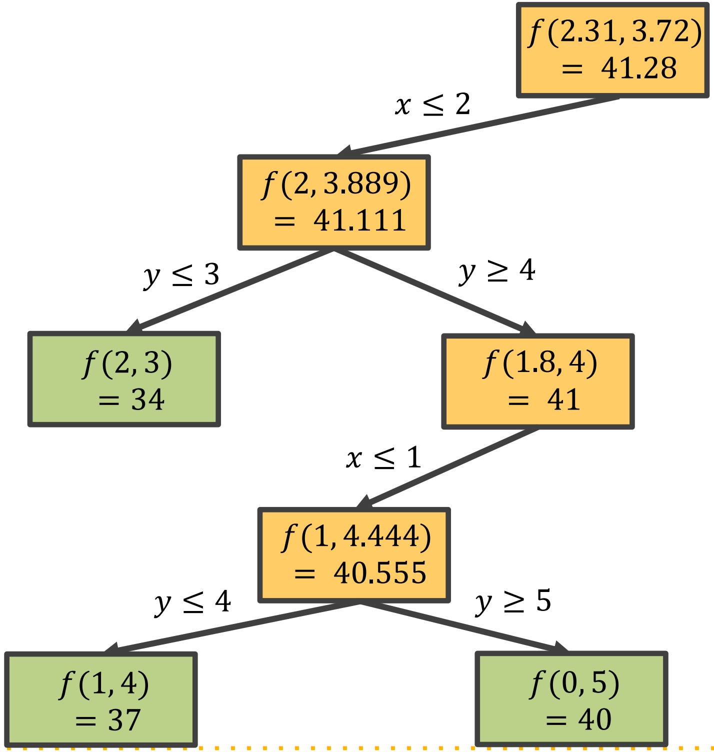




Example: Branch & Bound

max	$f(x, y) = 5x + 8y$
s.t.	$5x + 9y \leq 45$
	$1.1x + 1.2y \leq 7$
	$x, y \in [0..100]$





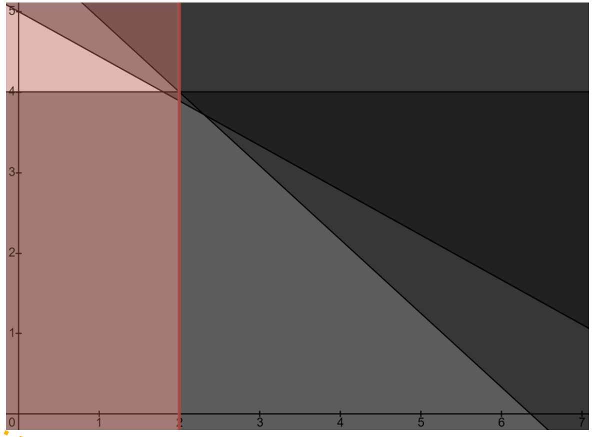
Example: Branch & Bound

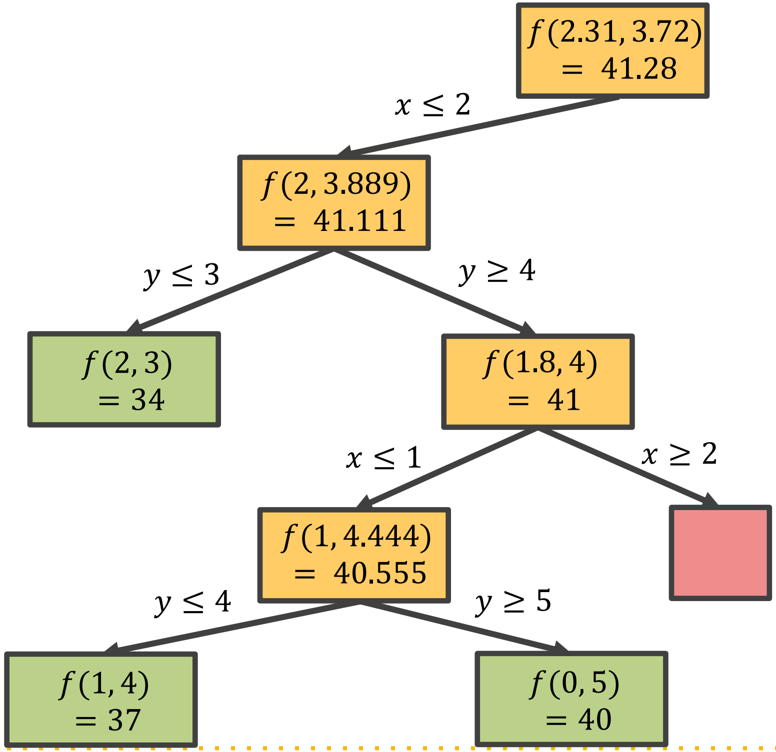
max $\quad f(x, y) = 5x + 8y$

s.t. $\quad 5x + 9y \leq 45$

$\quad 1.1x + 1.2y \leq 7$

$\quad x, y \in [0..100]$





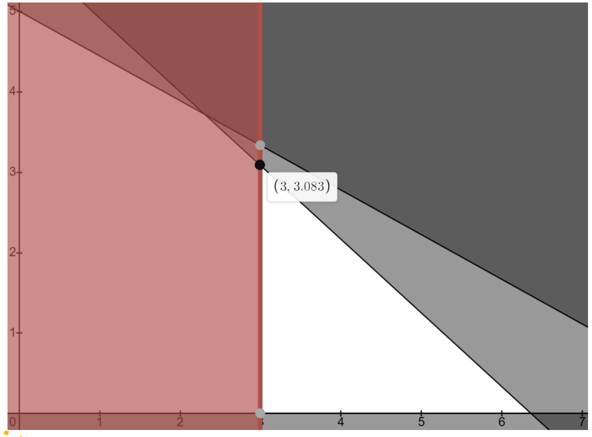
Example: Branch & Bound

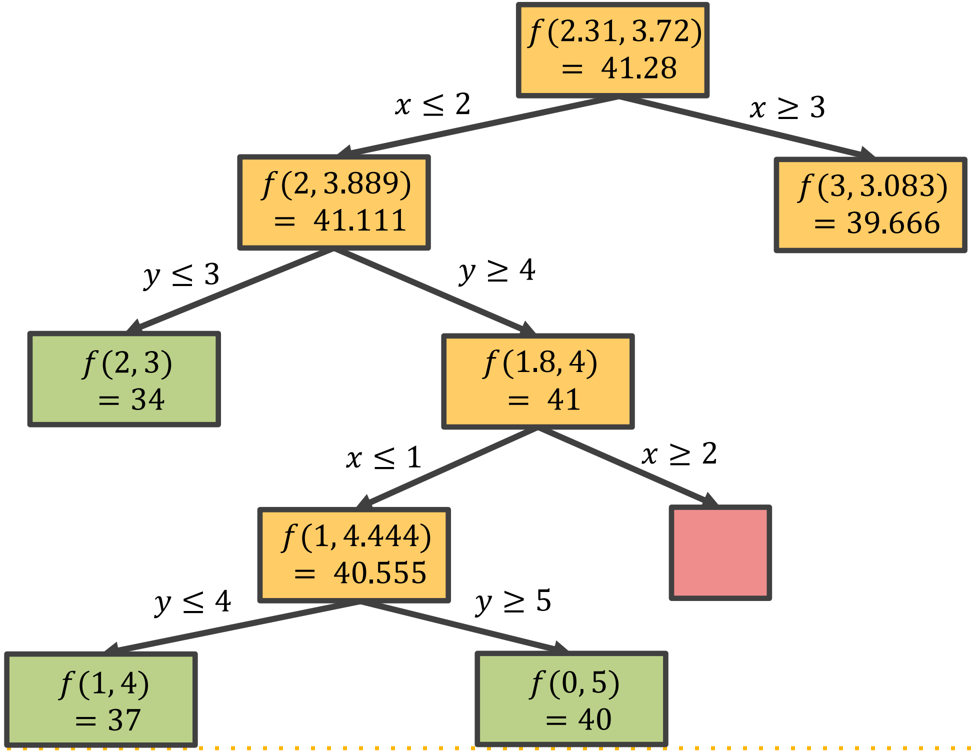
max $f(x, y) = 5x + 8y$

s.t. $5x + 9y \leq 45$

$1.1x + 1.2y \leq 7$

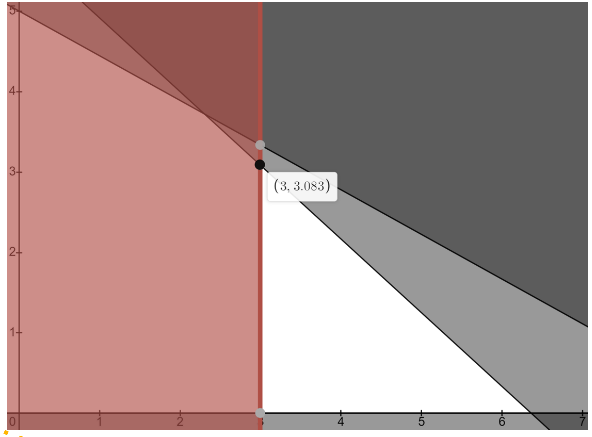
$x, y \in [0..100]$

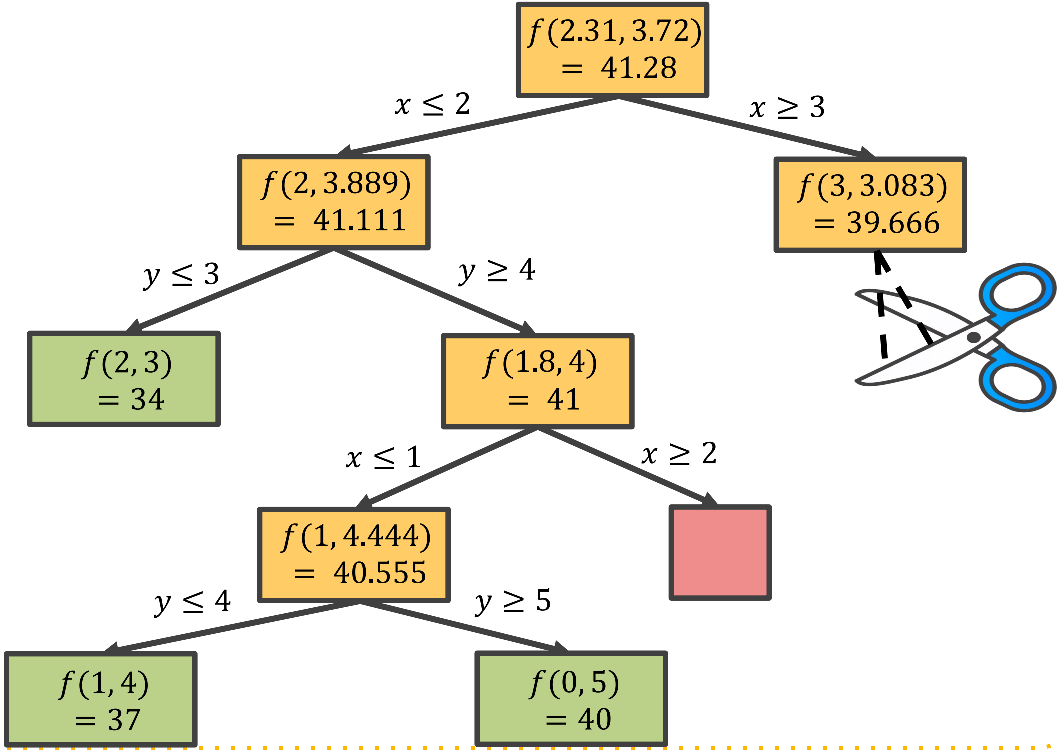




Example: Branch & Bound

max	$f(x, y) = 5x + 8y$
s.t.	$5x + 9y \leq 45$
	$1.1x + 1.2y \leq 7$
	$x, y \in [0..100]$





Iterative Branch & Bound

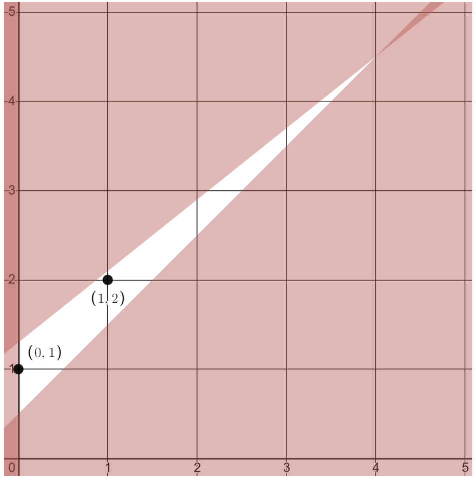
```
# find the optimal objective value for P₀
branch_and_bound(P₀):
  let best_seen = -inf
  let subproblems_to_visit = {P₀}
  while to_visit is nonempty:
    let P = subproblems_to_visit.pop()
    let LP_soln = solve_LP(LP(P))
    if LP_soln = INFEASIBLE: continue
    if objective_value(LP_soln) ≤ best_seen: continue
    if LP_soln satisfies integrality constraints for P:
      set best_seen = objective_value(LP_soln)
      continue
    let x be an int var with fractional value v in LP_soln
    subproblems_to_visit.add(branch_and_bound(P|x ≤ ⌊v⌋))
    subproblems_to_visit.add(branch_and_bound(P|x ≥ ⌈v⌉))
  return best_seen
```

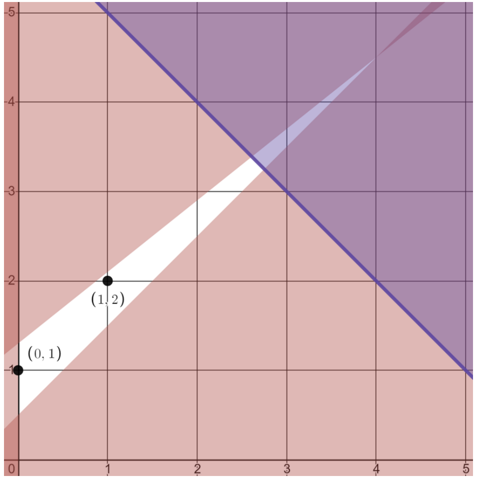
Tuning Branch & Bound

- What choices can we make when implementing branch and bound?
- Which subproblem to visit next?
  - Visit first-added subproblem (BFS)
  - Visit last-added subproblem (DFS)
  - Visit subproblem with best LP objective (“best-first search”)
- Which variable to branch on?
  - Most constrained variable (smallest domain, e.g. booleans)
  - Largest/smallest coefficient in objective function
  - Closest/farthest to halfway between integers (e.g. value of 0.5)
- Most solvers allow user to tune these based on knowledge of problem

Improving B&B with Cuts

- Informally, a **cut** for a MIP P is a new constraint (inequality) that doesn't eliminate any feasible solutions for P, but does for $LP(P)$
  - Tighter LP relaxation means we convergence faster to MIP solution!





Branch & Cut

- If we can find cuts of MIP, then add them and recurse on new MIP!
    - How to find cuts? Out of scope – method based on simplex algorithm
- Otherwise, branch to create subproblems as before

- Proposed by Manfred Padberg and Giovanni Rinaldi in 1989

The Knapsack Problem

- Given n items with values $v_1, \ldots, v_n$ and weights $w_1, \ldots w_n$, select maximum-value subset to fit into a knapsack with capacity W.

Fractional Knapsack

- What if items are subdivisible? Want to decide how much of each item to take (as a fraction from 0 to 1).
- Intuitively, do we want to prioritize... most valuable items? Lightest items? Something else?
- **Greedy algorithm:** Sort items by value-to-weight ratio. Take as much of each item as possible, in order, until knapsack is full.

0/1 Knapsack

- In the 0/1 knapsack problem, we either select an item or we don't.
- Does greedy algorithm still work?
  - No: 0/1 knapsack is NP-complete!
- Other (NP-complete) forms:
  - Multiple knapsacks
  - Multi-dimensional knapsack
  - Bin-packing

MIP for 0/1 Knapsack

- MIP formulation is very straightforward:

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i=1}^{n} x_i v_i \\ &\text{subject to} && \textstyle\sum_{i=1}^{n} x_i w_i \leq W \end{aligned}$$

- Why use MIP instead of…
  - $O(nW)$ dynamic programming algorithm
  - $O(n \lg n)$ approximation algorithm (at least 50% of optimal)

B&B for Knapsack

- How can we use branch and bound as an **algorithm paradigm** for the 0/1 knapsack problem (without using MIP)?

```
b&b_knapsack(items, W, best_seen):
    let fractional_soln = greedy_fractional(items, W)
    if value(fractional_soln) ≤ best_seen:
        return -inf
    if fractional_soln has no fractionally-selected items:
        return value(fractional_soln)
    let x be a fractionally-selected item in fractional_soln
    let obj1 = b&b_knapsack(items - {x}, W, best_seen)}
    set best_seen = max{obj1, best_seen}
    let obj2 = v(x) + b&b_knapsack(items - {x}, W - w(x), best_seen - v(x))
    return max{obj1, obj2}
```