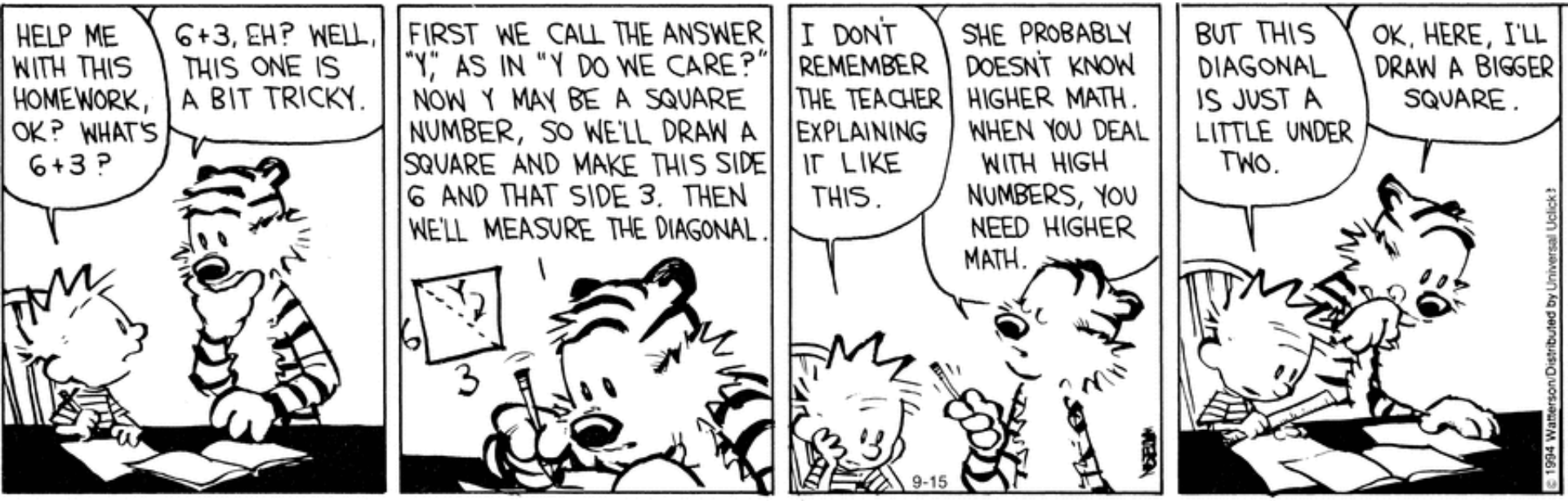
Automated Theorem Proving: DPLL and Simplex



One-Slide Summary

- An **automated theorem prover** is an algorithm that determines whether a mathematical or logical proposition is **valid** (**satisfiable**).
- A **satisfying** or **feasible assignment** maps variables to values that satisfy given constraints. A theorem prover typically produces a proof or a satisfying assignment (e.g., a counter-example backtrace).
- The **DPLL** algorithm uses efficient heuristics (involving "pure" or "unit" variables) to solve **Boolean Satisfiability** (SAT) quickly in practice.
- The **Simplex** algorithm uses efficient heuristics (involving visiting feasible corners) to solve **Linear Programming** (LP) quickly in practice.

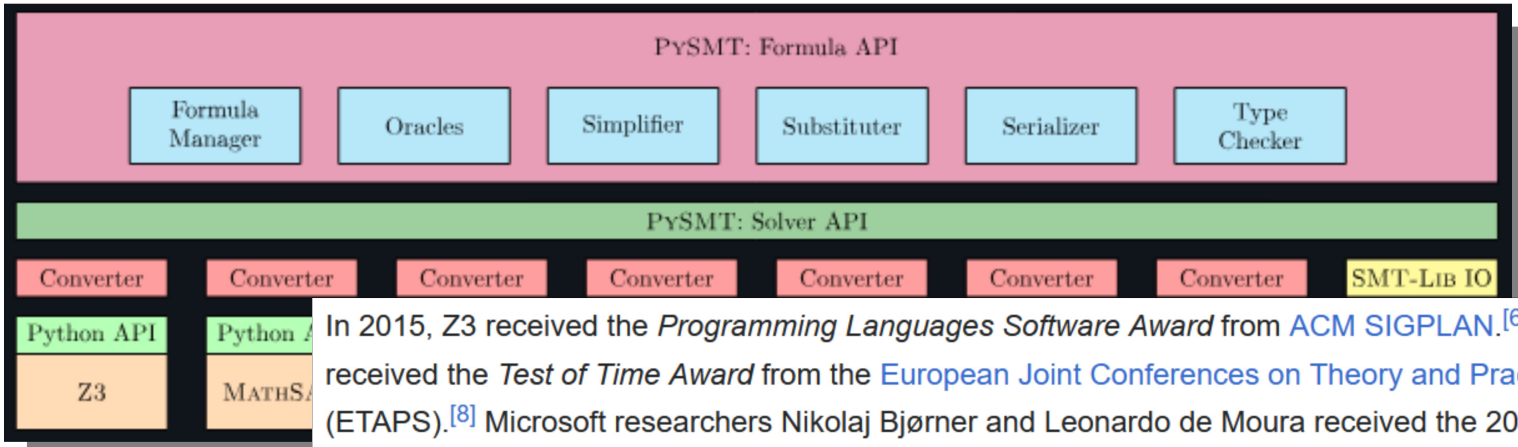
Why Bother?

- I am loathe to teach you anything that I think is a waste of your time.
- The use of “constraint solvers” or “SMT solvers” or “automated theorem provers” is endemic in PL, SE and Security research, among others.
- Many high-level analyses and transformations call Chaff, Z3 or Simplify (etc.) as a black box single step.
  - Opinion: PL tools call SMT the way “everyone else” uses LLMs.

Examples

Dafny is a **verification-ready programming language**. As you type in your program, Dafny's verifier constantly looks over your shoulder, flags any errors, shows you counterexamples, and congratulates you when your code matches your specifications. When you're done, Dafny can **compile your code to C#, Go, Python, Java, or JavaScript** (more to come!), so it can integrate with your existing workflow.

Dafny verifies your program using a type of theorem prover known as a Satisfiability Modulo Theories (SMT) solver. More specifically, it uses the Z3 solver. In many cases, it's possible to state only the final properties you want your program to have, with annotations such as `requires` and `ensures` clauses, and let the prover do the rest for you, automatically.



In 2015, Z3 received the *Programming Languages Software Award* from ACM SIGPLAN.[6][7] In 2018, Z3 received the *Test of Time Award* from the European Joint Conferences on Theory and Practice of Software (ETAPS).[8] Microsoft researchers Nikolaj Bjørner and Leonardo de Moura received the 2019 Herbrand Award for Distinguished Contributions to Automated Reasoning in recognition of their work in advancing theorem proving with Z3.[9][10]

Recent Examples (PLDI 2024)

- "**SMT solvers are foundational tools** for reasoning about constraints in practical problems both within and outside program analysis. Faster SMT solving improves the performance of practical tools and expands the set of tractable problems. … This paper introduces a theory arbitrage ..."
  - SMT Theory Arbitrage: Approximating Unbounded Constraints using Bounded Theories
- "Given an observed serializable execution of a data store application, IsoPredict generates and **solves SMT constraints** to find an unserializable execution that is a feasible execution of the application."
  - IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications
- "Value-flow bug finding can be implemented via a forward traversal on the graphs, during which the alias constraints and property-specific constraints can be gathered together and **handed to an SMT solver**."
  - Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis
- "Reachability can be proved by Symbolic Execution (SE) [Cadar and Sen, 2013], as shown on Algorithm 1. SE enumerates all paths □□and **converts them to SMT formulas** ..."
  - Quantitative Robustness for Vulnerability Assessment

Examples

- “VeriCon uses first-order logic to specify admissible network topologies and desired network-wide invariants, and then implements classical Floyd-Hoare-Dijkstra **deductive verification using Z3**.”
  - VeriCon: Towards Verifying Controller Programs in Software-Defined Networks, PLDI 2014
- “However, the search strategy is very different: our synthesizer fills in the holes using component-based synthesis (as opposed to **using SAT/SMT solvers**).”
  - Test-Driven Synthesis, PLDI 2014
- “If the terms *l*, *m*, and *r* were of type *nat*, this **theorem is solved automatically** using Isabelle/HOL's built-in *auto* tactic.”
  - Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain, PLDI 2014

Desired Examples

- SLAM
  - Given “new = old” and “new++”, can we conclude “new = old”?
  - $(new_0 = old_0) \wedge (new_1 = new_0 + 1) \wedge (old_1 = old_0) \Rightarrow (new_1 = old_1)$
- Division By Zero
  - IMP: “print x/((x*x)+1)”
  - $(n_1 = (x * x) + 1) \Rightarrow (n_1 \neq 0)$

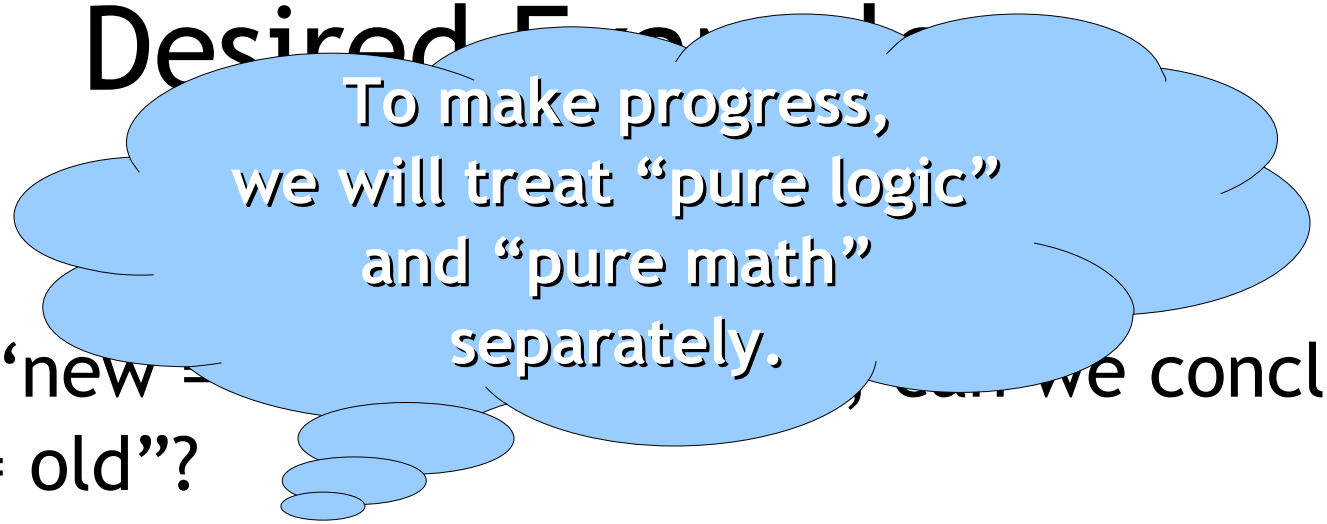
Incomplete

- Unfortunately, we can't have nice things.

- **Theorem (Godel, 1931)**. No consistent system of axioms whose theorems can be listed by an algorithm is capable of proving all truths about relations of the natural numbers.

- But we can profitably restrict attention to *some* relations about numbers.

Desired Examples



- SLAM
  - Given “new = [illegible] can we conclude “new = old”?
  - $(new_0 = old_0) \wedge (new_1 = new_0 + 1) \wedge (old_1 = old_0) \Rightarrow (new_1 = old_1)$
- Division By Zero
  - IMP: “print x/((x*x)+1)”
  - $(n_1 = (x * x) + 1) \Rightarrow (n_1 \neq 0)$

Overall Plan

- Satisfiability

- Simple SAT Solving
- Practical Heuristics
- DPLL algorithm for SAT

Logic

- Linear programming
- Graphical Interpretation
- Simplex algorithm

Math

Boolean Satisfiability

- Start by considering a simpler problem: propositions involving only **boolean** variables

  bexp := x
  | bexp ∧ bexp
  | bexp ∨ bexp
  | ¬ bexp
  | bexp ⇒ bexp
  | true | false

- Given a bexp, return a satisfying assignment or indicate that it cannot be satisfied

Satisfying Assignment

- A **satisfying assignment** maps boolean variables to boolean values.
- Suppose $\sigma(x)$ = true and $\sigma(y)$ = false
- $\sigma \models x$ // $\models$ is "models" or "makes
- $\sigma \models x \vee y$ // true" or "satisfies"
- $\sigma \models y \Rightarrow \neg x$
- $\sigma \not\models x \Rightarrow (x \Rightarrow y)$
- $\sigma \not\models \neg x \vee y$

Cook-Levin Theorem

- **Theorem (Cook-Levin). The boolean satisfiability problem is NP-complete.**
- In '71, Cook published “The complexity of theorem proving procedures”. Karp followed up in '72 with “Reducibility among combinatorial problems”.
  - Cook and Karp received Turing Awards.
- SAT is in NP: verify the satisfying assignment
- SAT is NP-Hard: we can build a boolean expression that is satisfiable iff a given nondeterministic Turing machine accepts its given input in polynomial time

Conjunctive Normal Form

- Let's make it easier (but still NP-Complete)
- A **literal** is "variable" or "negated variable"

  x $\quad\quad$ $\neg y$

- A **clause** is a disjunction of literals

  $(x \vee y \vee \neg z)$ $\quad\quad$ $(\neg x)$

- **Conjunctive normal form** (CNF) is a conjunction of clauses

  $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (z)$

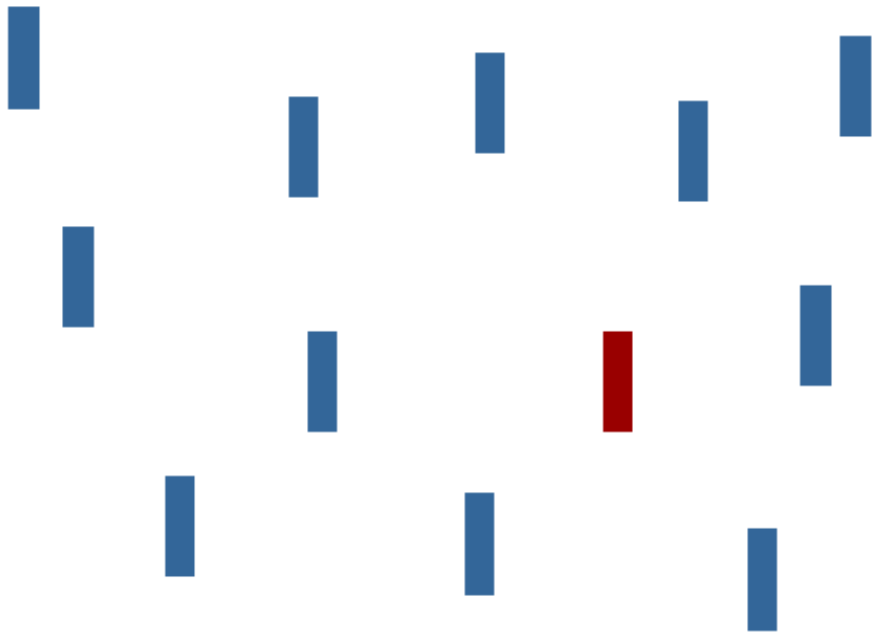
- Must satisfy all clauses at once
  - "global" constraints!

SAT Solving Algorithms

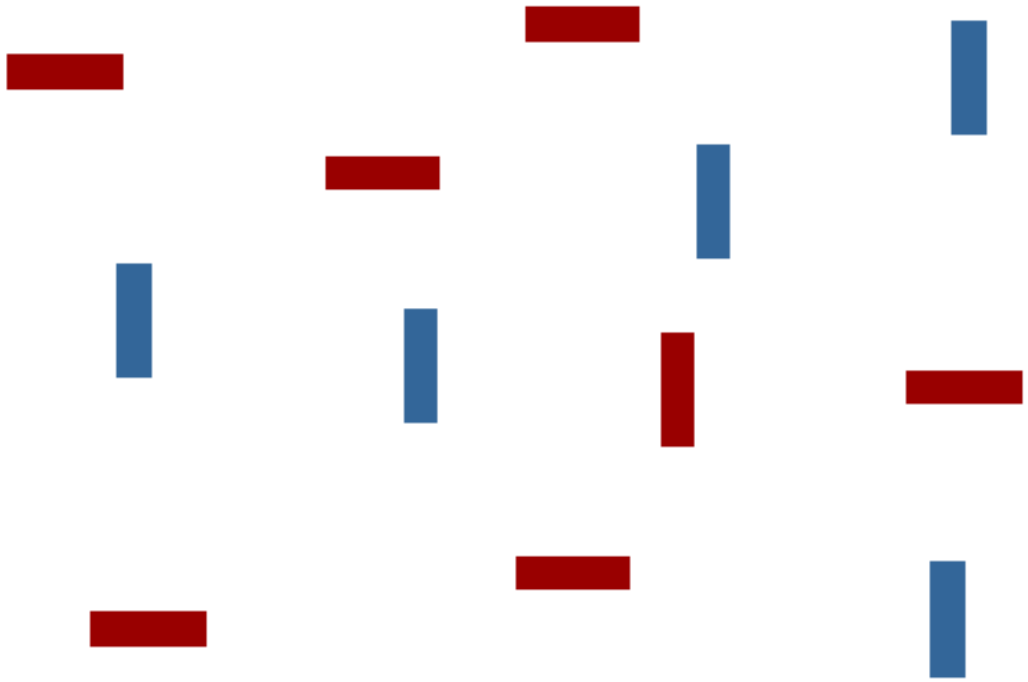
$$\exists \sigma.\ \sigma \vDash (x \vee y \vee \neg z) \ \wedge \ (\neg x \vee \neg y) \ \wedge \ (z)$$

- So how do we solve it?
- Ex: $\sigma(x) = \sigma(z) = \text{true}, \sigma(y) = \text{false}$
- Expected running time?

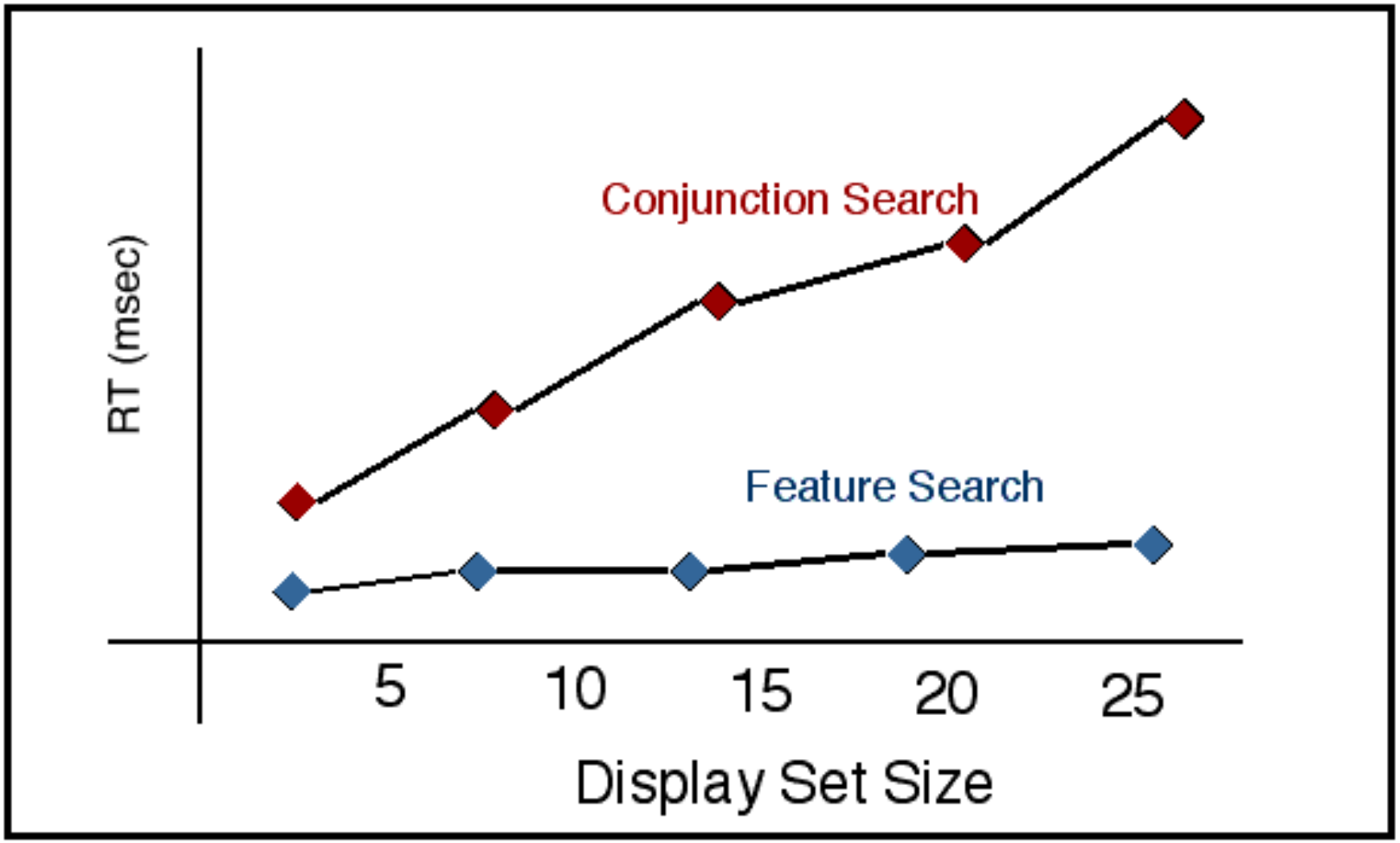
Analogy: Human Visual Search
“Find The Red Vertical Bar”



Human Visual Search
“Find The Red Vertical Bar”



Some Visual Features Admit O(1) Detection



Strangers On A Train

- https://www.youtube.com/watch?v=_tVFwhoeQVM

Think Fast: Partial Answer?

$$
\begin{aligned}
&(\neg a \vee \neg b \vee \neg c \vee d \vee e \vee \neg f \vee g \vee \neg h \vee \neg i) \\
\wedge\ &(\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee \neg g \vee h \vee \neg i) \\
\wedge\ &(a \vee \neg b \vee \neg c \vee \neg d \vee e \vee \neg f \vee \neg g \vee \neg h \vee i) \\
\wedge\ &(\neg b) \\
\wedge\ &(a \vee \neg b \vee c \vee \neg d \vee e \vee \neg f \vee \neg g \vee \neg h \vee i) \\
\wedge\ &(\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee \neg g \vee h \vee \neg i)
\end{aligned}
$$

- If this instance is satisfiable, what *must* part of the satisfying assignment be?

Think Fast: Partial Answer?

$$(\neg a \vee \neg b \vee \neg c \vee d \vee e \vee \neg f \vee g \vee \neg h \vee \neg i)$$
$$\wedge\ (\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee \neg g \vee h \vee \neg i)$$
$$\wedge\ (a \vee \neg b \vee \neg c \vee \neg d \vee e \vee \neg f \vee \neg g \vee \neg h \vee i)$$
$$\wedge\ (\neg b)$$
$$\wedge\ (a \vee \neg b \vee c \vee \neg d \vee e \vee \neg f \vee \neg g \vee \neg h \vee i)$$
$$\wedge\ (\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee \neg g \vee h \vee \neg i)$$

- If this instance is satisfiable, what *must* part of the satisfying assignment be? **b = false**

Need For Speed 2

$$
\begin{aligned}
&(\neg a \vee c \vee \neg d \vee e \vee f \vee \neg g \vee \neg h \vee \neg i) \\
\wedge\ &(\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee g \vee h \vee i) \\
\wedge\ &(\neg a \vee \neg b \vee c \vee e \vee f \vee g \vee \neg h \vee i) \\
\wedge\ &(\neg a \vee b \vee c \vee d \vee e \vee \neg f \vee \neg g \vee h \vee \neg i) \\
\wedge\ &\quad (b \vee \neg c \vee \neg d \vee e \vee \neg f \vee g \vee h \vee \neg i) \\
\wedge\ &(\neg a \vee b \vee c \vee d \vee \neg g \vee \neg h \vee \neg i)
\end{aligned}
$$

- If this instance is satisfiable, what *must* part of the satisfying assignment be?

Need For Speed 2

$$
\begin{aligned}
&(\neg a \vee c \vee \neg d \vee e \vee f \vee \neg g \vee \neg h \vee \neg i) \\
\wedge\ &(\neg a \vee b \vee \neg c \vee d \vee \neg e \vee f \vee g \vee h \vee i) \\
\wedge\ &(\neg a \vee \neg b \vee c \vee e \vee f \vee g \vee \neg h \vee i) \\
\wedge\ &(\neg a \vee b \vee c \vee d \vee e \vee \neg f \vee \neg g \vee h \vee \neg i) \\
\wedge\ &\quad (b \vee \neg c \vee \neg d \vee e \vee \neg f \vee g \vee h \vee \neg i) \\
\wedge\ &(\neg a \vee b \vee c \vee d \vee \neg g \vee \neg h \vee \neg i)
\end{aligned}
$$

- If this instance is satisfiable, what *must* part of the satisfying assignment be? **a = false**

Unit and Pure

- A **unit clause** contains only a single literal.
  - Ex: (x) (¬y)
  - Can only be satisfied by making that literal true.
  - There is no choice: just add it to the answer!
- A **pure variable** is either "always ¬ negated" or "never ¬ negated".
  - Ex: (¬x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y) ∧ (z)
  - Can only be satisfied by making that literal true.
  - There is no choice: just add it to the answer!

Unit Propagation

- If X is a literal in a unit clause, add X to that satisfying assignment and replace X with "true" in the input, then simplify:
  1. $(\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg z) \wedge (z)$
  2. identify "z" as a unit clause
  3. σ += "z = true"

Unit Propagation

- If X is a literal in a unit clause, add X to that satisfying assignment and replace X with "true" in the input, then simplify:
  1. $(\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg z) \wedge (z)$
  2. identify "z" as a unit clause
  3. σ += "z = true"
  4. $(\neg x \vee y \vee \neg \text{true}) \wedge (\neg x \vee \neg \text{true}) \wedge (\text{true})$

Unit Propagation

- If X is a literal in a unit clause, add X to that satisfying assignment and replace X with "true" in the input, then simplify:
  1. $(\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg z) \wedge (z)$
  2. identify "z" as a unit clause
  3. σ += "z = true"
  4. $(\neg x \vee y \vee \neg \text{true}) \wedge (\neg x \vee \neg \text{true}) \wedge (\text{true})$
  5. $(\neg x \vee y) \wedge (\neg x)$
- Profit! Let's keep going ...

Unit Propagation FTW

5. $(\neg x \vee y) \quad \wedge \quad (\neg x)$
6. Identify "$\neg x$" as a unit clause
7. σ += "$\neg x$ = true"
8. $(\text{true} \vee y) \quad \wedge \quad (\text{true})$
9. done!

$\{z, \neg x\} \vDash (\neg x \vee y \vee \neg z) \ \wedge \ (\neg x \text{ or } \neg z) \ \wedge \ (z)$

Pure Variable Elimination

- If V is a variable that is always used with one polarity, add it to the satisfying assignment and replace V with “true”, then simplify.
  1. $(\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$
  2. identify “$\neg y$” as a pure literal

Pure Variable Elimination

- If V is a variable that is always used with one polarity, add it to the satisfying assignment and replace V with “true”, then simplify.
  1. $(\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$
  2. identify “$\neg y$” as a pure literal
  3. $(\neg x \vee \text{true} \vee \neg z) \wedge (x \vee \text{true} \vee z)$
  4. Done.

DPLL

- The **Davis-Putnam-Logemann-Loveland** (DPLL) algorithm is a complete decision procedure for CNF SAT based on:
  - Identify and propagate *unit* clauses
  - Identify and propagate *pure* literals
  - If all else fails, exhaustive *backtracking* search
- It builds up a partial satisfying assignment over time.

DP '60: “A Computing Procedure for Quantification Theory”

DLL '62: “A Machine Program for Theorem Proving”

DPLL Algorithm

```
let rec dpll (c : CNF) (σ : model) : model option =
  if σ ⊨ c then                          (* polytime to check *)
    return Some(σ)                       (* we win! *)
  else if ( ) in c then                  (* empty clause *)
    return None                          (* unsat *)
  let u = unit_clauses_of c in
  let c, σ = fold unit_propagate (c, σ) u in
  let p = pure_literals_of c in
  let c, σ = fold pure_literal_elim (c, σ) p in
  let x = choose ((literals_of c) - (literals_of σ)) in
  return (dpll (c ∧ x) σ) or (dpll (c ∧ ¬x) σ)
```

DPLL Example

$(x \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (w) \wedge (w \vee y)$

- Unit clauses: (w)

  $(x \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

- Pure literals: $\neg y$

  $(x \vee \neg z)$

- Choose unassigned: x (recursive call)

  $(x \vee \neg z) \wedge (x)$

- Unit clauses: (x)
- Done! $\sigma=\{w, \neg y, x\}$

SAT Conclusion

- DPLL is commonly used by award-winning SAT solvers such as Chaff and MiniSAT
- Not explained here: how you "choose" an unassigned literal for the recursive call
  - This "branching literal" is the subject of many papers on heuristics
- Performance matters. Example: specialize a MiniSAT solver to a particular problem class

  Justyna Petke, Mark Harman, William B. Langdon, Westley Weimer: **Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class.** European Conference on Genetic Programming (EuroGP) 2014 (silver human competitive award)

Japanese Literature

- This 11th-century Japanese work is often regarded as the world's first novel. It was written by Murasaki Shikibu, a Heian noblewoman. A psychological and historical work, it details the life and romantic adventures of a “shining” prince. It features over 400 characters and a strong internal consistency (e.g., they all age at the same time and follow feudal and family relationships).

Sports & Recreation

- Thus juggling, circus or recreational object is spun using a string attached to two hand sticks. Modern constructions are derived from the Chinese (空钟 or 扯铃) via Europe. The modern European name is said to derive from its older name, “the devil on two sticks". Advanced tricks include the whip catch, integral and finger grind.

Q: Computer Science

- This American mathematician and scientist developed the simplex algorithm for solving linear programming problems. In 1939 he arrived late to a graduate stats class at UC Berkeley where Professor Neyman had written two famously unsolved problems on the board. The student thought the problems “seemed a little harder than usual” but a few days later handed in complete solutions, believing them to be homework problems overdue. This real-life story inspired the introductory scene in *Good Will Hunting*.

Linear Programming

- Example Goal:
  - Find X such that $X > 5 \wedge X < 10 \wedge 2X = 16$
- Let $x_1 \ldots x_n$ be real-valued variables
- A satisfying assignment (or **feasible solution**) is a mapping from variables to reals satisfying all available constraints
- Given a set of linear constraints and a linear objective function to maximize, **Linear Programming** (LP) finds a feasibile solution that maximizes the objective function.

Linear Programming Instance

- Maximize $c_1x_1 + c_2x_2 + \ldots + c_nx_n$
- Subject to $a_{11}x_1 + a_{12}x_2 + \ldots \leq b_1$

  $a_{21}x_1 + a_{22}x_2 + \ldots \leq b_2$

  $a_{n1}x_1 + a_{n2}x_2 + \ldots \leq b_n$

  $x_1 \geq 0, \ldots, x_n \geq 0$

- Don't “need” the objective function
- Don't “need” $x_1 \geq 0$

2D Running Example

- Maximize $4x + 3y$
- Subject to

$$2x + 3y \leq 6 \quad (1)$$

$$2y \leq 5 \quad (2)$$

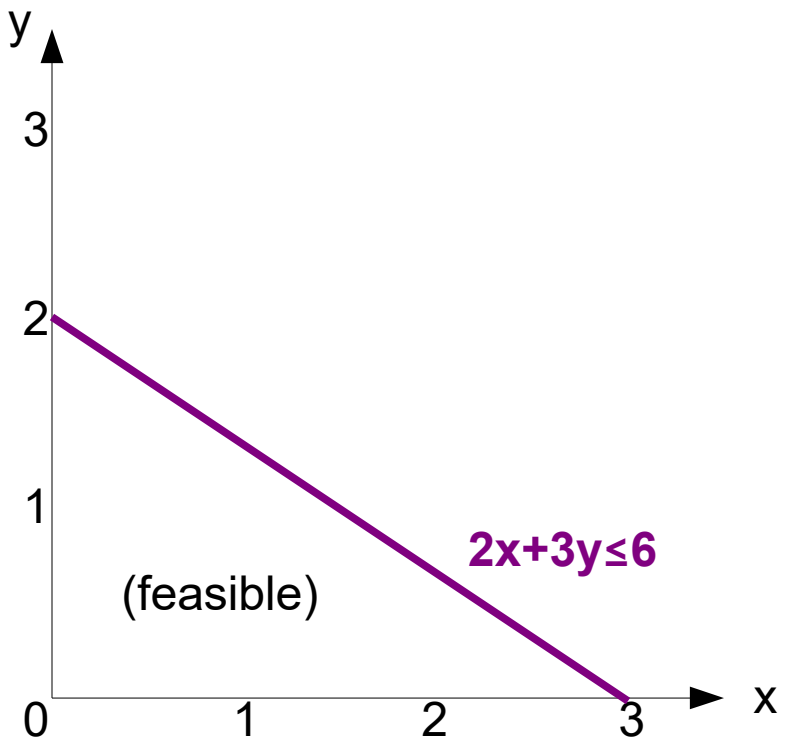
$$2x + 1y \leq 4 \quad (3)$$

$$x \geq 0,\ y \geq 0$$

- Feasible: (1,1) or (0,0)
- Infeasible: (1,-1) or (1,2)

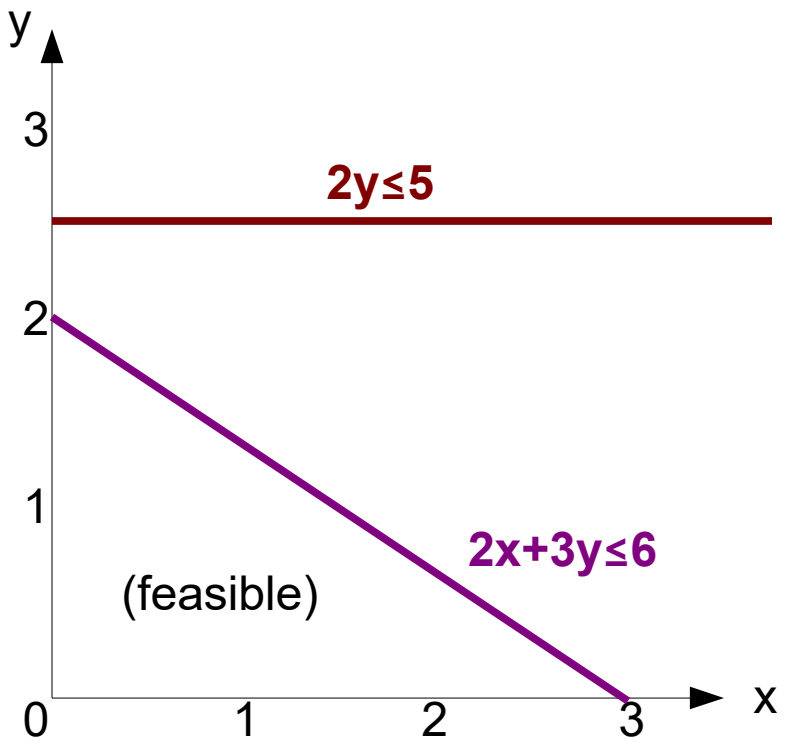
Key Insight

- Each linear constraint (e.g., $2x+3y \leq 6$) corresponds to a **half-plane**
  - A feasible half-plane and an infeasible one

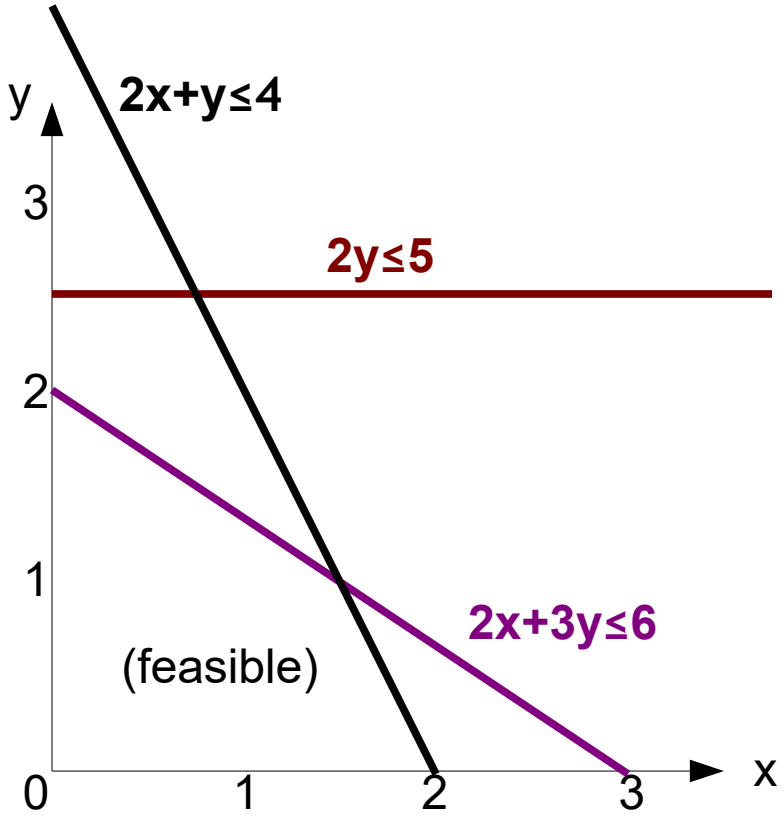


Key Insight

- Each linear constraint (e.g., $2y \leq 5$) corresponds to a **half-plane**
  - A feasible half-plane and an infeasible one



Key Insight

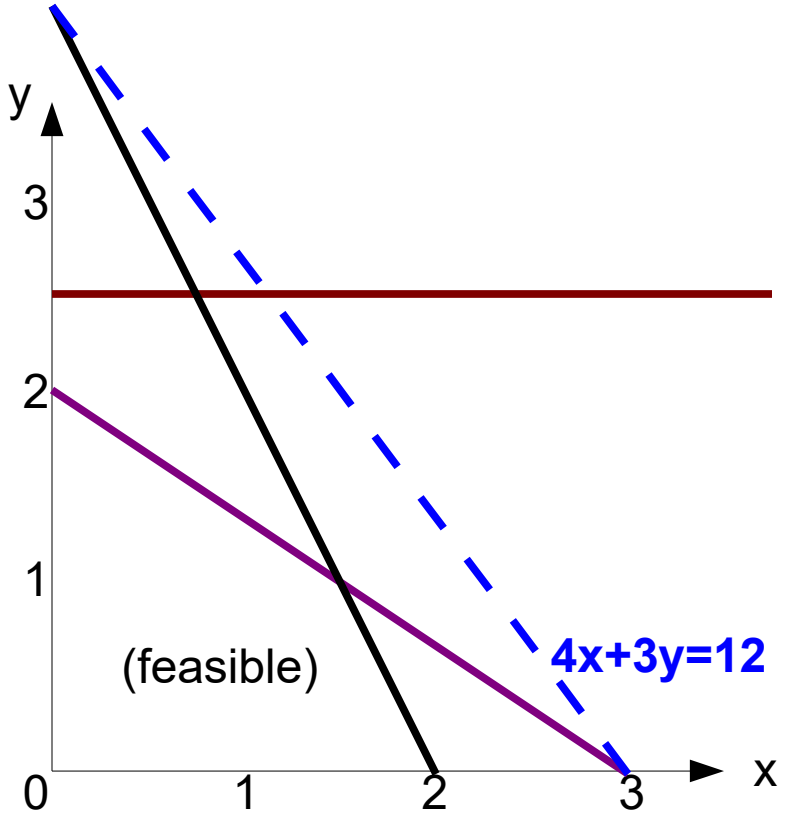


Feasible Region

- The region that is on the "correct" side of all of the lines is the **feasible region**
- If non-empty, it is always a **convex** polygon
  - Convex, for our purposes: if A and B are points in a convex set, then the points on the line segment between A and B are also in that convex set
- Optimality: "Maximize $4x + 3y$"
- For any c, $4x+3y=c$ is a line with the same slope
- **Corner points** of the feasible region must maximize
  - Why? Linear objective function + convex polygon

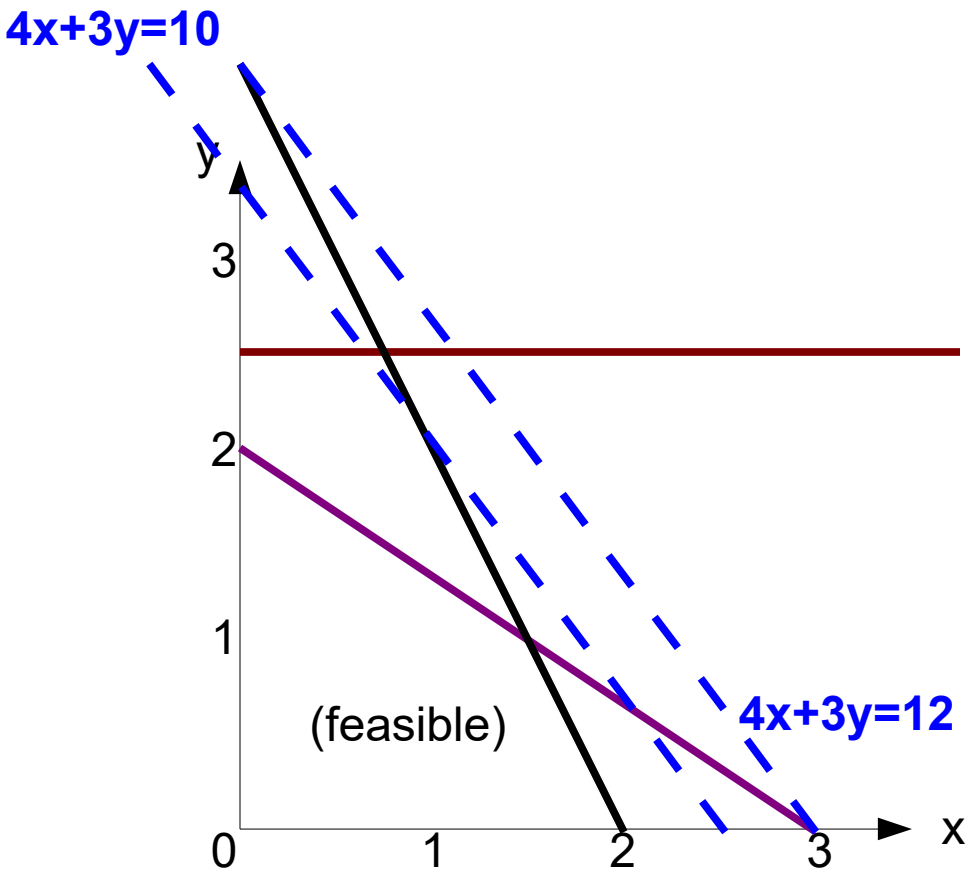
Objective Function

- Maximize 4x+3y



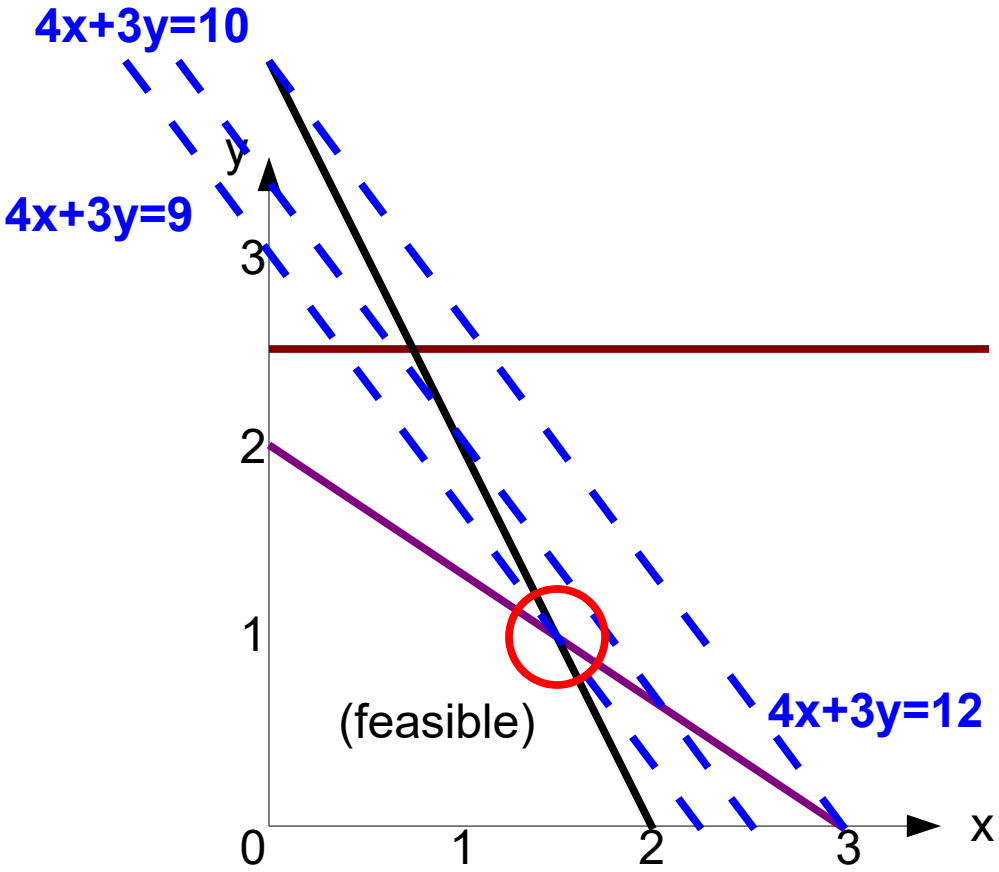
Objective Function

- Maximize 4x+3y



Objective Function

- Maximize 4x+3y

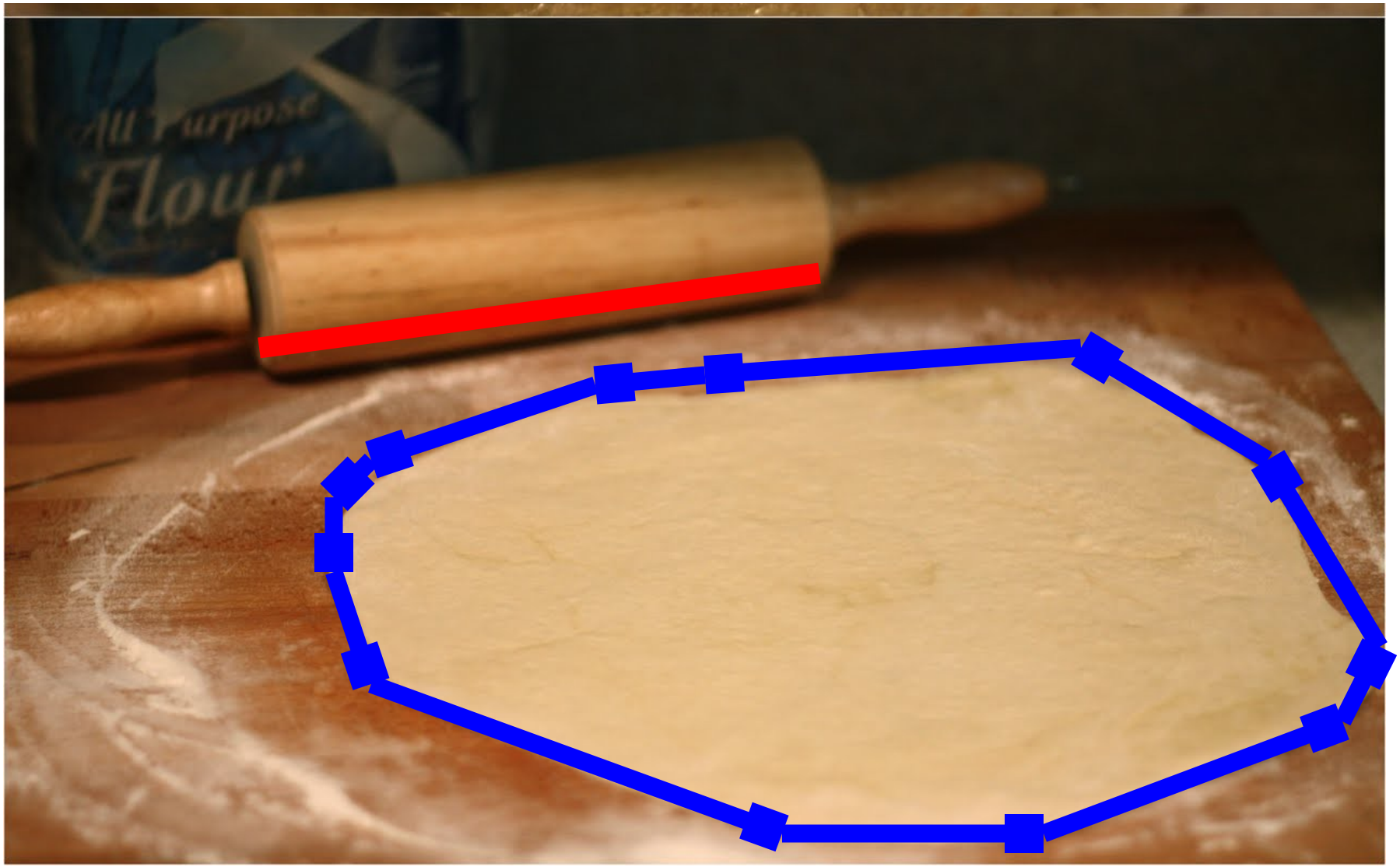


Optimal Corner Point (1.5, 1)
It's the feasible point that maximizes the objective function!

Analogy: Rolling Pin, Pizza Dough

Analogy: Rolling Pin, Pizza Dough

Analogy: Rolling Pin, Pizza Dough



Any Convex Pizza and Any Linear Rolling Pin Approach

Any Convex Pizza and Any Linear Rolling Pin Approach

Linear Programming Solver

- Three Step Process
  - Identify the coordinates of all feasible corners
  - Evaluate the objective function at each one
  - Return one that maximizes the objective function
- This totally works! We're done.

- The trick: how can we find all of the coordinates of the corners *without* drawing the picture of the graph?

Finding Corner Points

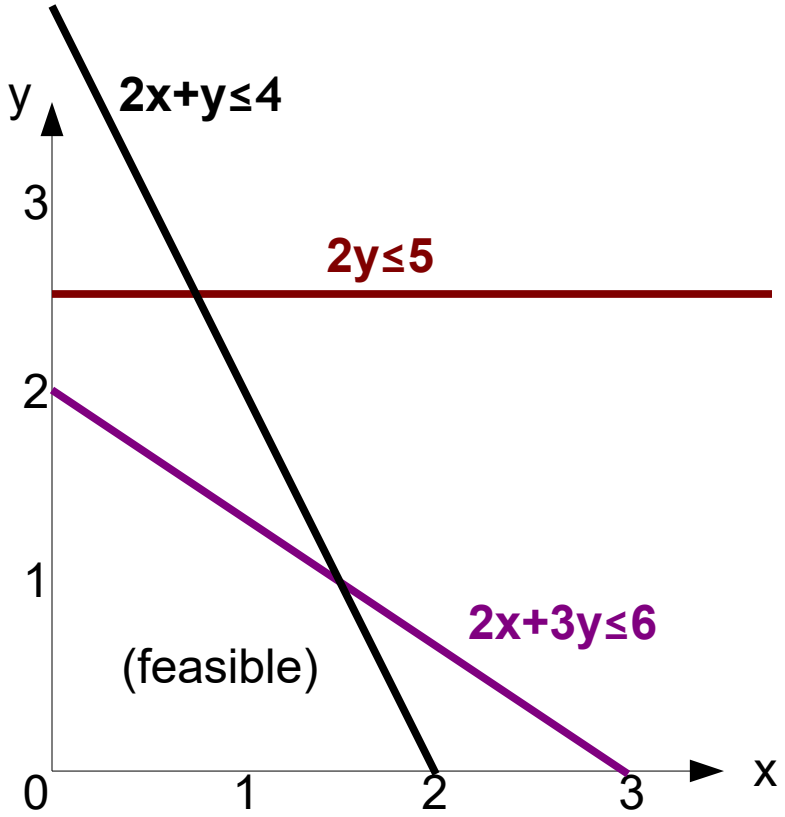
- A **corner point** (**extreme point**) lies at the intersection of constraints.
- Recall our running example:
- Subject to

$$\begin{aligned} 2x + 3y &\le 6 \quad (1)\\ 2y &\le 5 \quad (2)\\ 2x + 1y &\le 4 \quad (3)\\ x \ge 0,\ y &\ge 0 \end{aligned}$$

- Take just *(1)* and *(3)* as **defining equations**

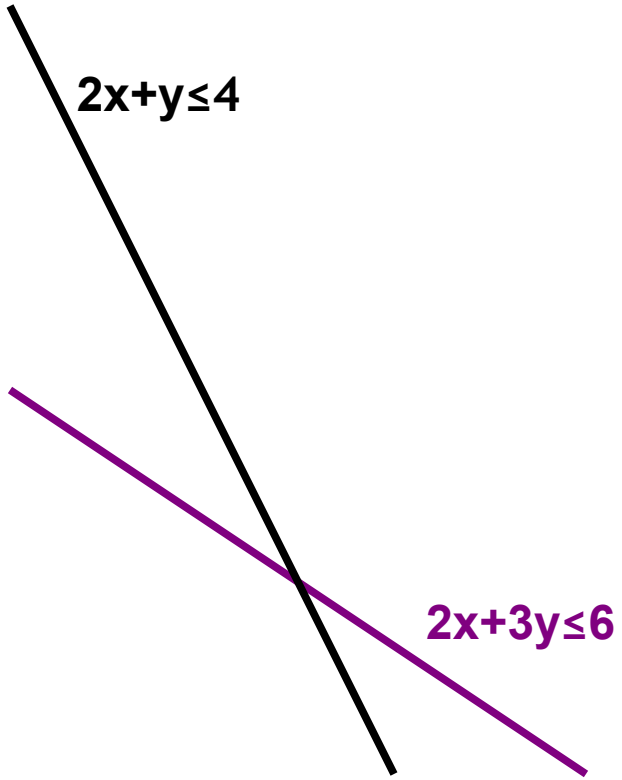
Visually

- $2x + 3y \leq 6$ and $2x + 1y \leq 4$
  - Hard to see with the whole graph ...



Visually

- $2x + 3y \leq 6$ and $2x + 1y \leq 4$
  - But easy if we only look at those two!

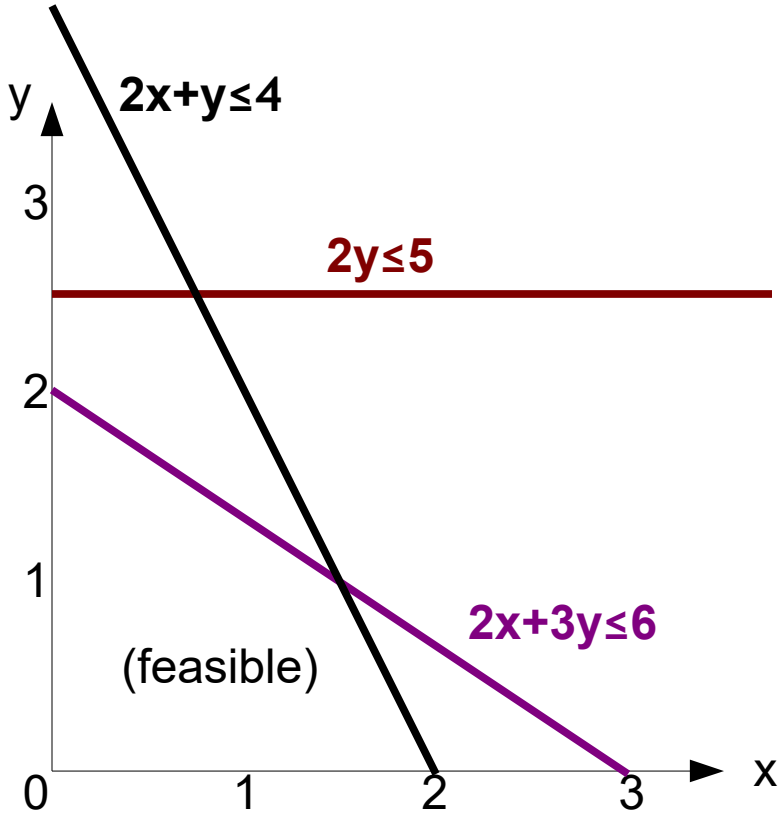


Mathematically

- $2x + 3y \leq 6$
- $2x + 1y \leq 4$
- Recall linear algebra: **Gaussian Elimination**
  - Subtract the second row from the first
- $0x + 2y \leq 2$
  - Yields "y = 1"
- Substitute "y=1" back in
- $2x + 3 \leq 6$
  - Yields "x = 1.5"

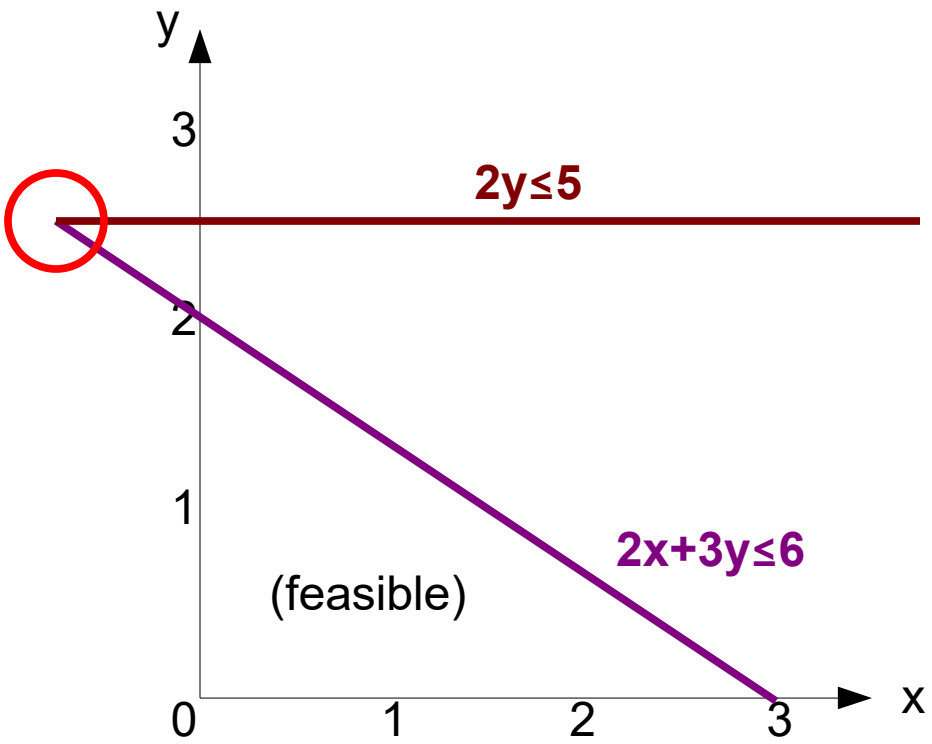
Infeasible Corners

- $2x + 3y \leq 6$ and $2y \leq 5$



Infeasible Corners

- $2x + 3y \leq 6$ and $2y \leq 5$
  - (-0.75,2.5) solves the equations but it does not satisfy our "$x \geq 0$" constraint: infeasible!



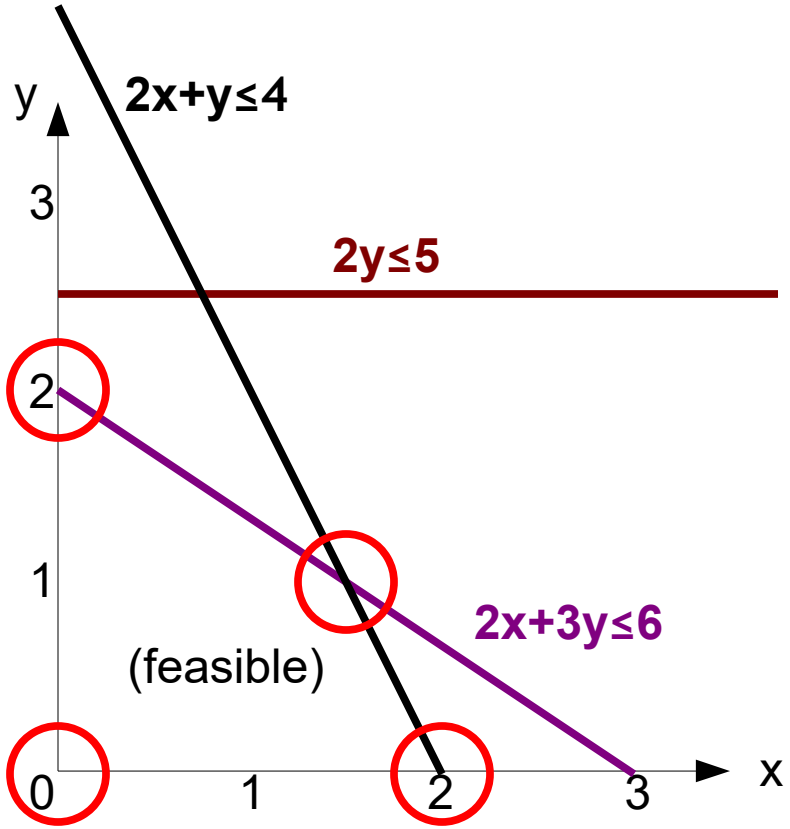
Solving Linear Programming

- Identify the coordinates of all corners
  - Consider all pairs of constraints, solve each pair
  - Filter to retain points satisfying all constraints
- Evaluate the objective function at each point
- Return the point that maximizes

- With 5 equations, the number of pairs is “6 choose 2” = 5!/(2!3!) = 10.
  - Only 4 of those 10 are feasible.

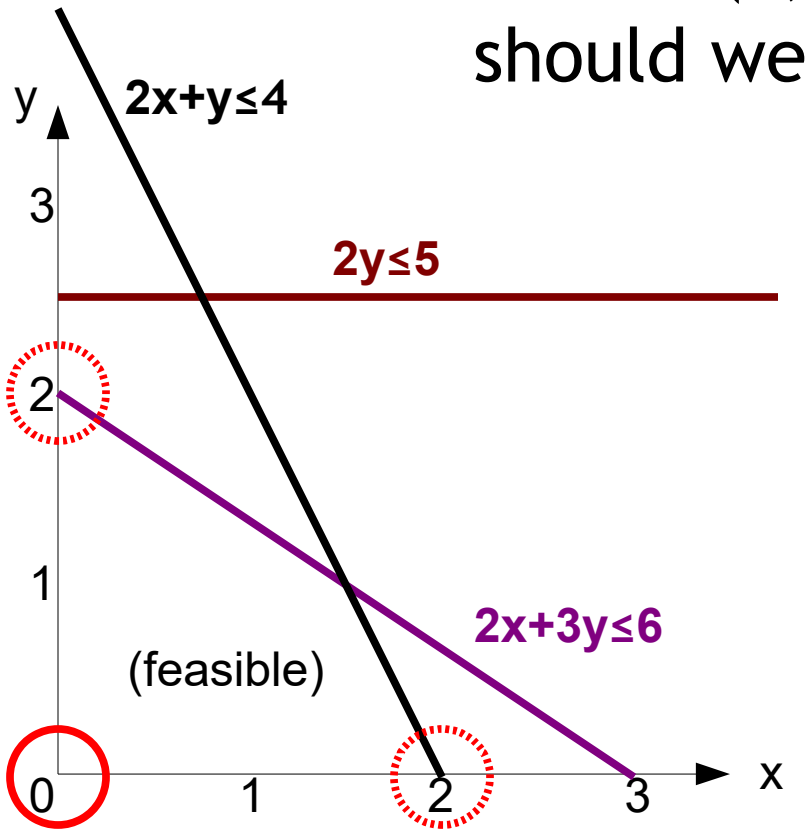
Feasible Corners

- In our running example, there are four feasible corners



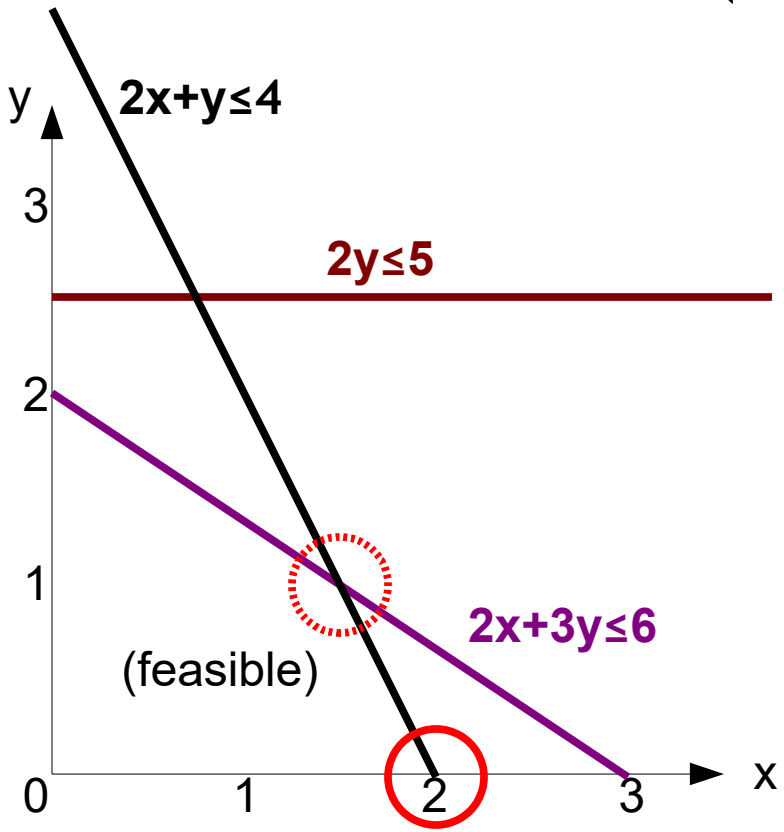
Road Trip!

- Suppose we start in one feasible corner (0,0)
  - And we know our objective function 4x+3y
  - Do we move to corner (0,2) or (2,0) next, or should we stay here?



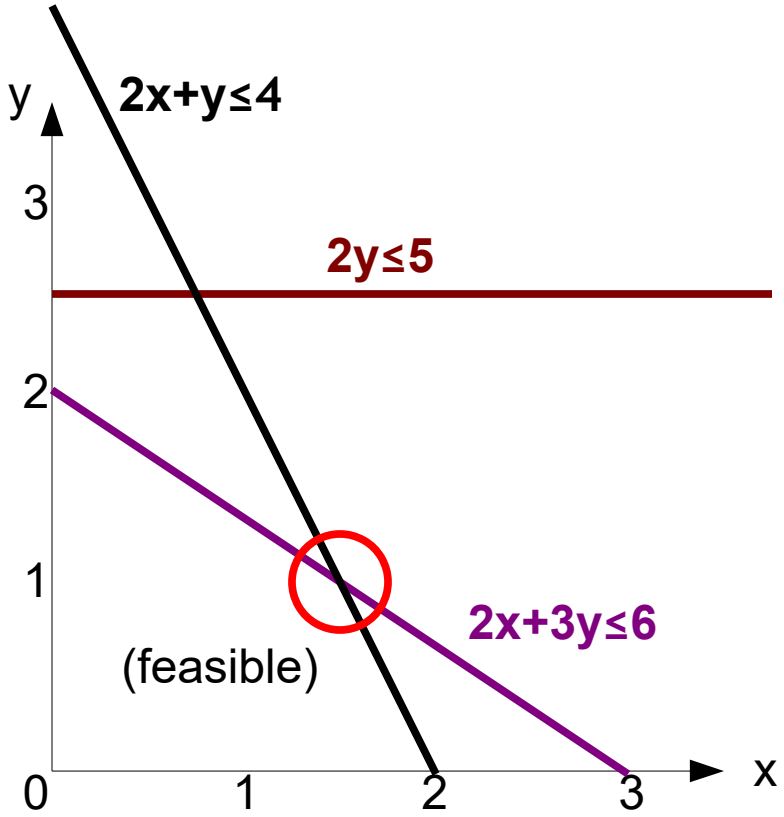
Road Trip!

- We're now in (2,0)
  - And we know our objective function 4x+3y
  - Do we move to corner (1.5,1) or stay here?



Road Trip!

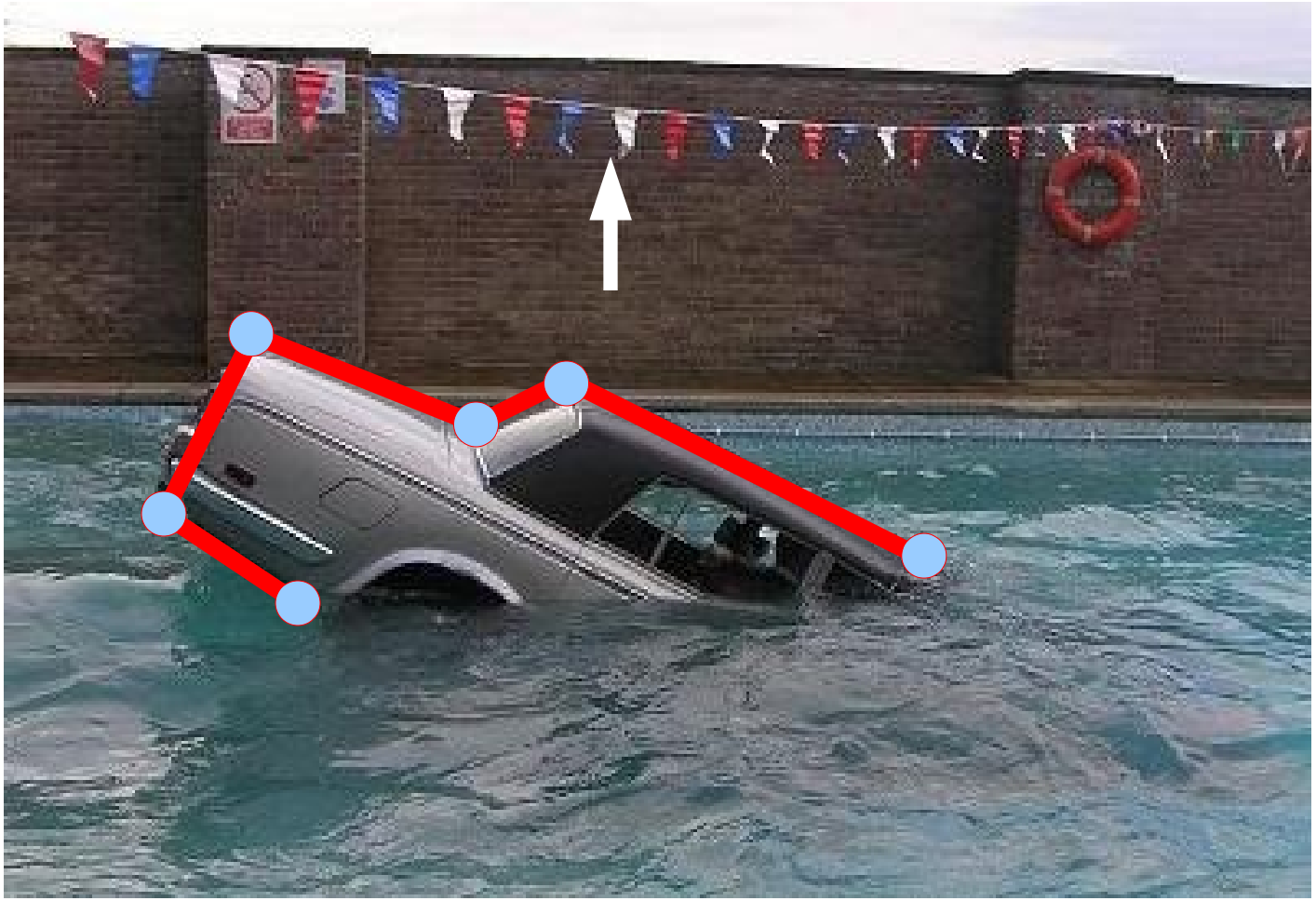
- We're now in (1.5,1)
  - We're done! We have considered all of our neighbors and we're the best.



Analogy: Don't Sink!

Reach Highest Point Greedily

Not A Counter-Example
Why Not?



Simplex Insight

- The **Simplex** algorithm encodes this “gradient ascent” insight: if there are many corners, we may not even need to enumerate or visit them all.
- Instead, just walk from feasible corner to adjacent feasible corner, maximizing the objective function every time.
  - It's linear and convex: you can't be “tricked” into a local maximum that's not also global.
- In a high-dimensional case, this is a huge win because there are many corners.

Simplex Algorithm

- **George Dantzig** published the Simplex algorithm in 1947.
  - John von Neumann theory prize, US National Medal of Science, "one of the top 10 algorithms of the 20th century", etc.
- Phase 1: find any feasible corner
  - Ex: solve two constraints until you find one
- Phase 2: walk to best adjacent corner
  - Ex: "pivot" row operations between the "leaving" variable and the "entering" variable
- Repeat until no adjacent corner is better

Simplex Running Time

(special thanks to Yi Tang)

- Linear programming (via the interior-point method, ellipsoid algorithm) can be solved in worst-case polynomial-time.
  - n variables encoded in L input bits: $O(n^6L)$ time
  - Open question: is there a strongly polytime algorithm for linear programming over the reals?
- Simplex is quite efficient in practice.
  - In a formal sense, “most” LP instances can be solved by Simplex in polytime. “Hard” instances are “not dense” in the set of all instances (akin to: the Integers are “not dense” in the Reals).
- 0-1 Integer Linear Programming is NP-Hard.

Next Time

- DPLL(T) combines DPLL + Simplex into one grand unified theorem prover

Homework

- HW2 Due Soon
- Reading for next time