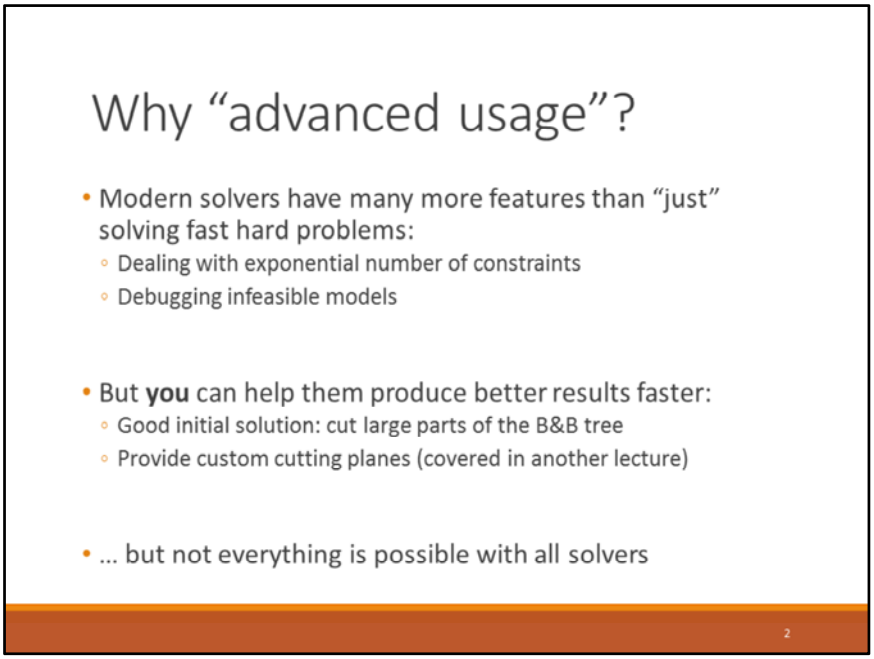
Advanced solver usage

For julia and JuMP

THIBAUT CUVELIER

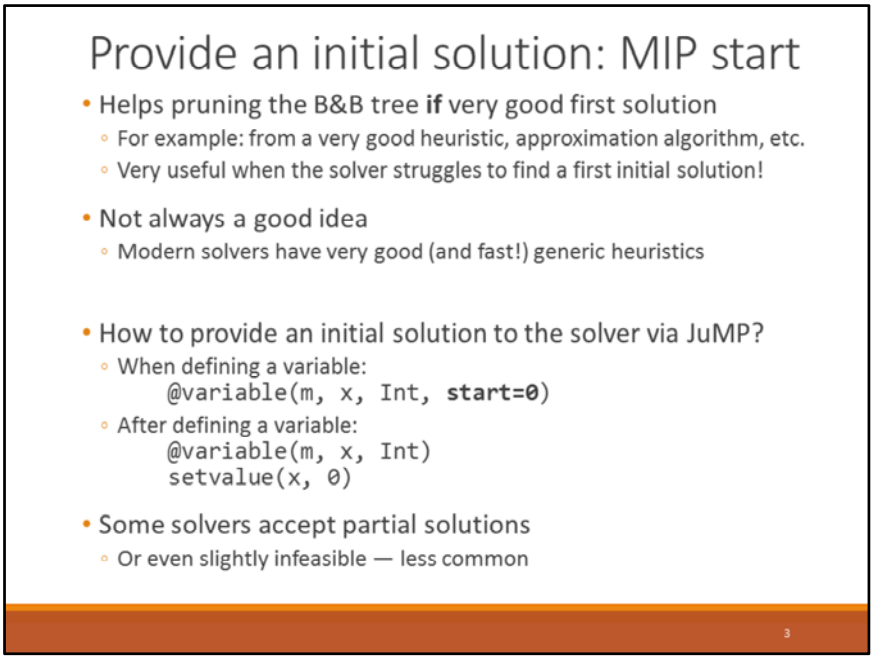
23 SEPTEMBER, 2016

1



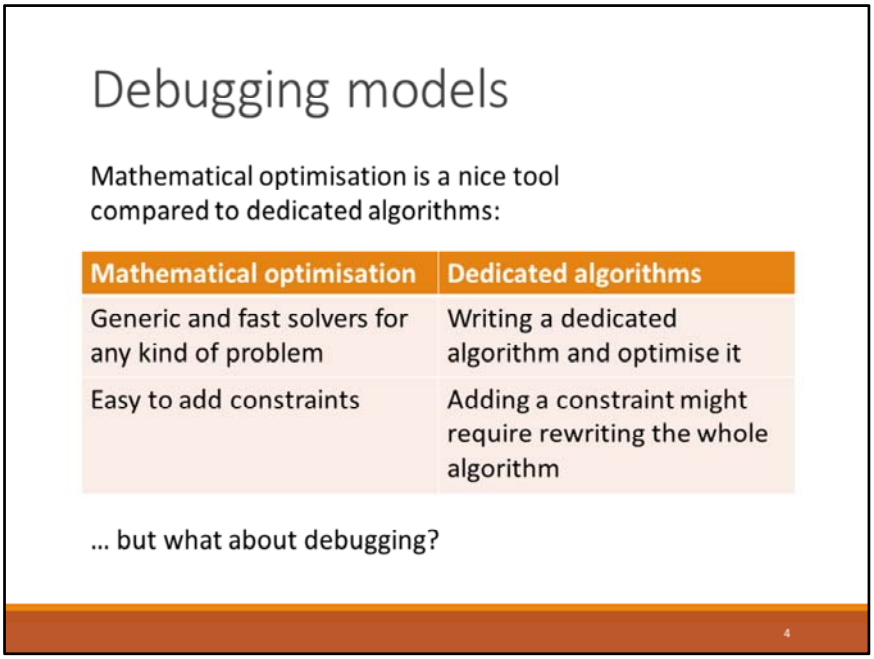
Exponential constraints: helps you (less complicated code to write), helps the solver (not too many constraints to handle).
Not all features available in all solvers: Cbc, Mosek have no lazy constraints (albeit Mosek is commercial); Cbc has no way to detect infeasibility.

Orthogonal to better formulations (comparison): a better formulation gives a better LP relaxation, hence the root relaxation is closer to an integer solution and has a better objective value (comparable with a good initial point: cuts large parts of the B&B tree).

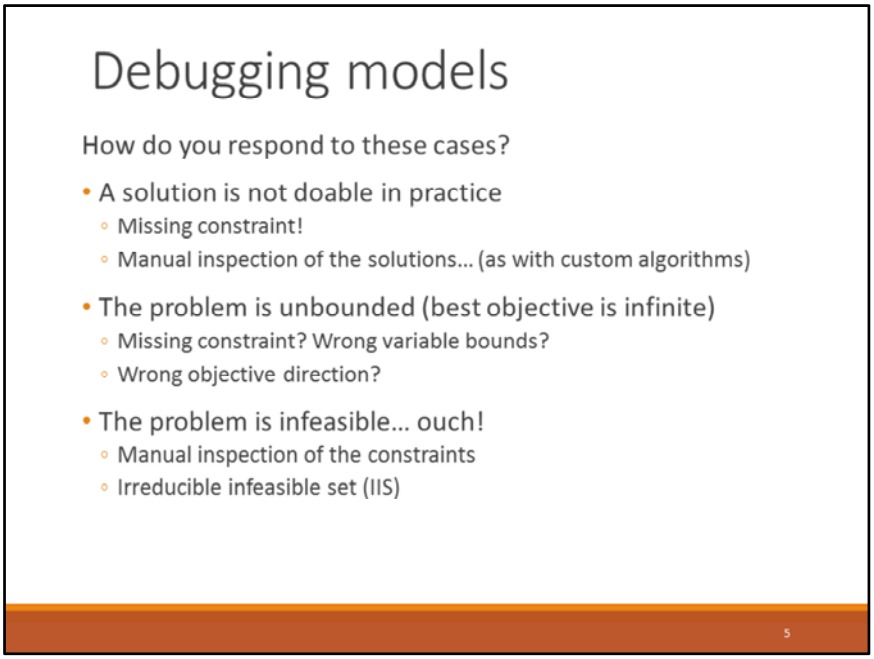


Effect on the B&B tree? Example for minimisation: a good initial solution is feasible and has a low objective value. In the B&B tree, when branching, if a subproblem gives a larger objective value, then the corresponding branch does not need to be explored: it cannot yield better solutions that the one already known. If the solution is very good, the solver might also decide to directly try to prove it is optimal — rather than going deep in the tree to find good integer-feasible solutions.

What if a solution is partial? The solver will use heuristics to complete it. Gurobi and CPLEX do it.
What if it is slightly infeasible? The solver might use reparation heuristics to respect all constraints. CPLEX does it, Gurobi not.



Debugging: you know how to do it for algorithms, but your current tools do not work for MIP models (declarative programming instead of imperative).



I.e. real-world constraints, those that are translated into linear constraints (for example, the project statements present many real-world constraints). For example, you try to produce two things on the same machine when it cannot do more than one.

Debugging models: IIS

- The IIS is (one of) the smallest set(s) of constraints that make(s) the problem infeasible
- Focus on analysing those constraints to solve the infeasibility
- Still need a lot of work on the user's side…
  but helps a lot: focus on a few constraints rather than the full model
- Not directly available within JuMP…

Debugging models: IIS

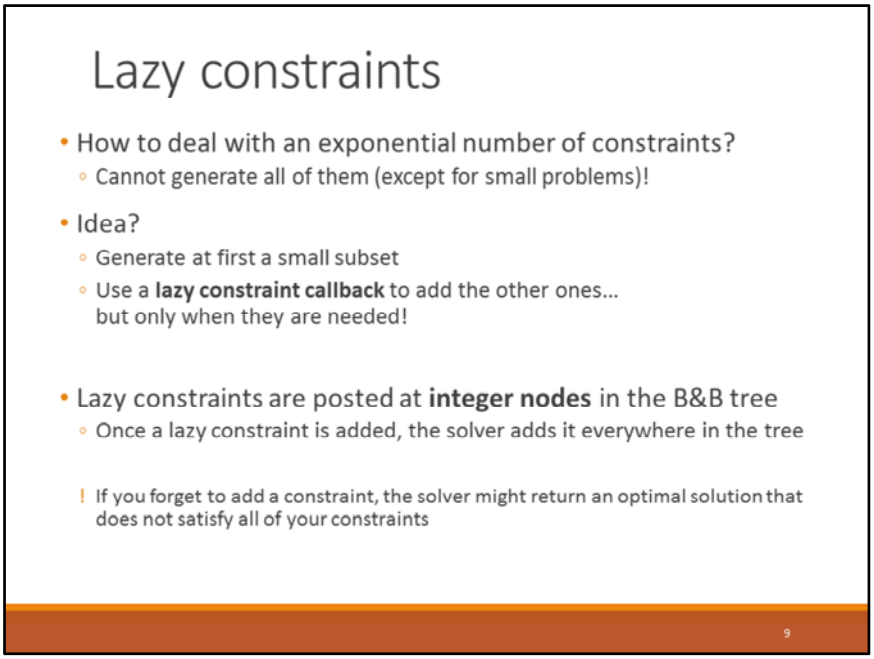
- Low-level access to the IIS computed by Gurobi:

```
grb = MathProgBase.getrawsolver(internalmodel(m))
Gurobi.computeIIS(grb)
Gurobi.get_intattrelement(grb, "IISConstr", 3)
# Is constraint 3 in IIS?
```

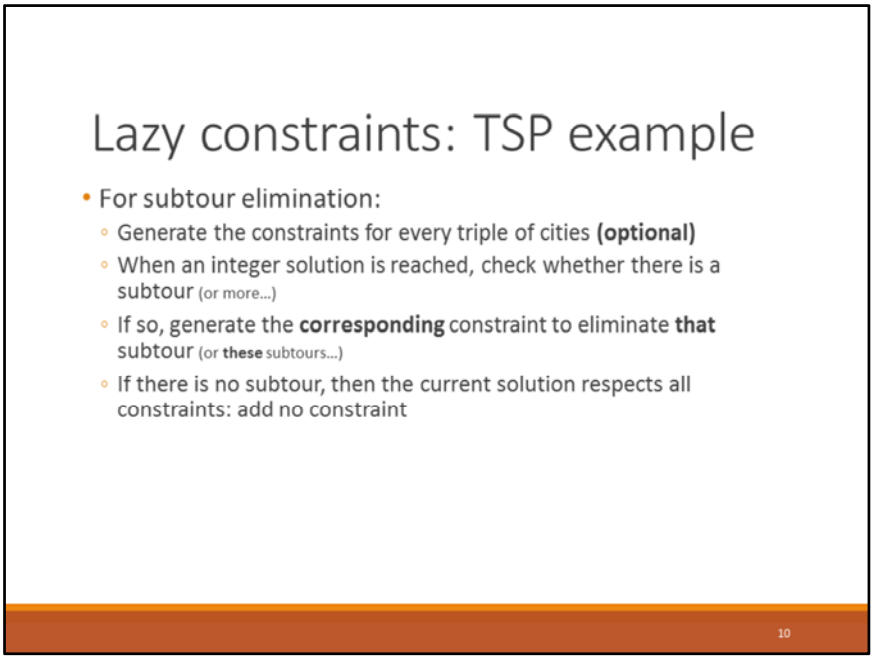
Debugging models: IIS

- Better way: use Gurobi's shell
  - But less integrated!
- Methodology:
  - From Julia:
    - Export your model into a LP file: `writeLP(m, "m.lp")`
  - Run Gurobi's Python shell:
    - Load your model: `m = read("m.lp")`
    - Optimise this model: `m.optimize()`
    - Compute the IIS: `m.computeIIS()`
    - Export it (ILP file): `m.write("m.ilp")`

8



Those lazy constraints are valid throughout within the B&B tree, even though they are added when they are required at a specific node.
The solver does the required bookkeeping about the previously found integer solutions, it adapts the B&B tree as required for the new constraint (some branches become infeasible, e.g.). It might use reparation heuristics to get already found integer solutions feasible for the new lazy constraints.



Could generate more than one subtour elimination constraint per callback call (i.e. multiple new constraints for one B&B node), but this is not required: forbidding one subtour will force the solver to look for another solution, where the other subtours are not necessarily present. If they are, then the corresponding constraints will be added, of course; but otherwise these extraneous constraints are not guaranteed to be useful.

Lazy constraints: TSP example

- With Gurobi: must indicate you will use lazy constraints!

```
m = Model(solver=GurobiSolver(LazyConstraints=1))
… # Model
```

- Write a separation function: find a subtour in the current solution

```
function findSubtour(sol)
  … # Separation procedure
end
```

- Write the actual callback: add a **useful** subtour elimination constraint

```
function subtour(cb)
  st, slength = findSubtour(getvalue(…))
  @lazyconstraint(cb, edges(st) <= slength - 1)
end
```

- Register the callback function and solve

```
addlazycallback(m, subtour)
solve(m)
```

11

Here: summary of the code to use lazy constraints, global architecture.

Lazy constraints: performance?

- If the separation procedure is **efficient**, then no need to spend time to generate all constraints
- How many constraints before attempting to solve?
  - Should be more efficient to generate those constraints than to let the callback add them later on
  - High likelihood of being useful (e.g., small subtours)
- "Useful" missing constraint?
  - **Any** missing constraint that is violated by the current solution
  - In some cases, use the **most infeasible** missing constraint (see later in the course)

Other kinds of callbacks?

- Solvers allow more callbacks than those lazy constraints:
  - User cuts: add valid inequalities, at any node in the B&B tree (see later)
    With JuMP: `addcutcallback` to register the function
    `@usercut` to add the cut
  - User heuristics: at an integer node, improve the current solution
    With JuMP: `addheuristiccallback` to register the function
    `setsolutionvalue` to indicate the value for a variable
    `addsolution` to post the heuristic solution
  - Informational callbacks: retrieve information from the solver, at any node
    With JuMP: `addinfocallback` to register the function

13