Workshop on constraint programming

13 Dec 2011

by
Srikumar Subramanian and Martin Henz
for students of CS1010R,
School of Computing
National University of Singapore

Errors, bugs, etc. are entirely mine. If you find them, please post to https://github.com/srikumarks/FD.js/issues.
-Srikumar Subramanian

Goals

* Intro to constraint programming over finite integer domains
* ... using fd.js Javascript library
* Misc supporting tools like fossil and the debugger.

Constraint programming

* "Variables" declared to take on values in an integer domain.
* A "constraint" serves to narrow the domains of related variables.
* "Propagators" are processes that implement constraints by watching for changes to domains.

Finite domain variables

Finite integer domain =

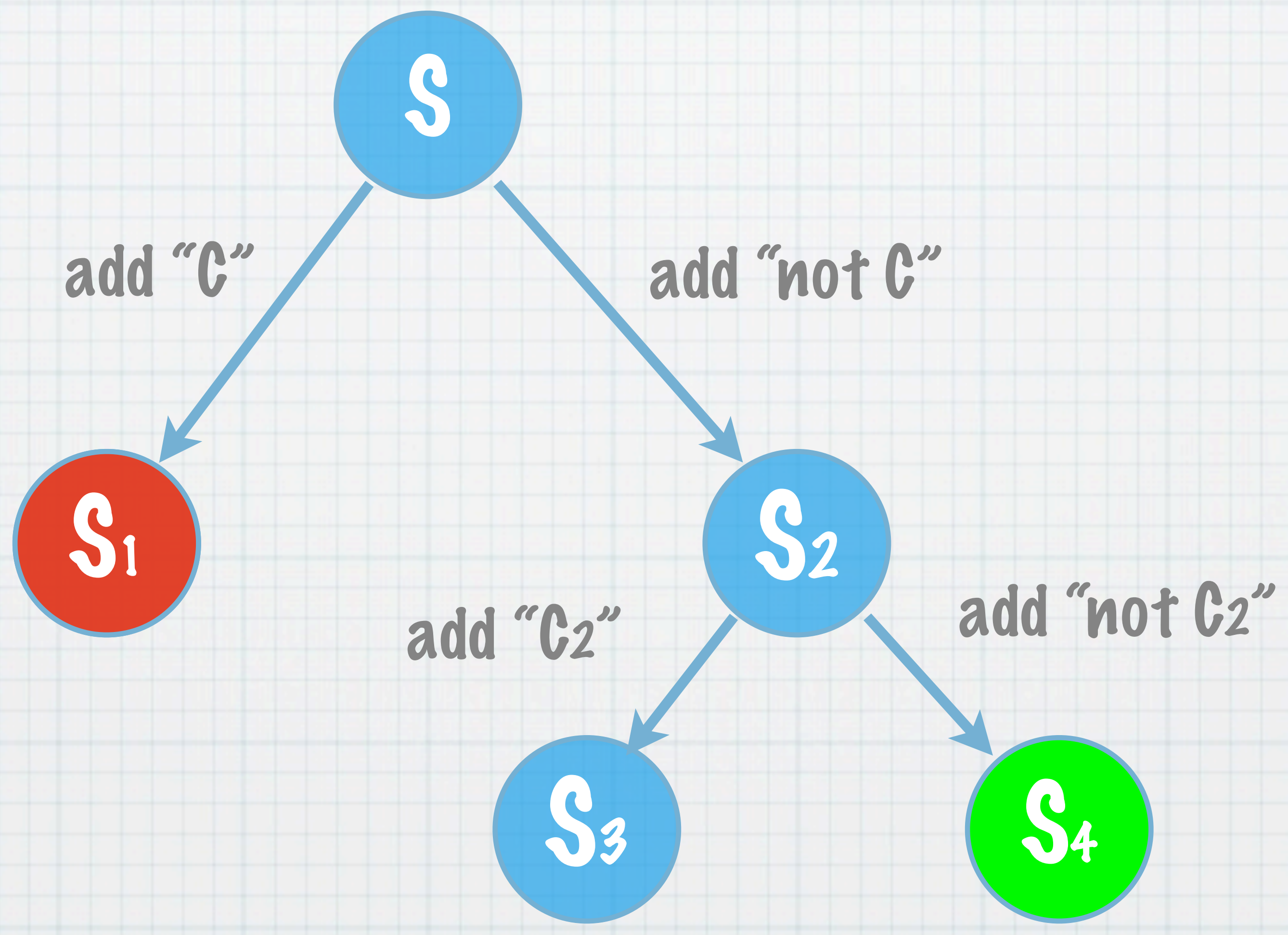
finite subset of Z,
usual range = 0 to a large number.

Common domain representation =

union of non-overlapping intervals
ex: `[[1,10],[15,57],[200,355]]`
(both bounds included)

Branching/Searching

When propagators can no longer narrow domains ...



To search for a solution to a constraint problem, you start with a "space" that holds finite domain variables and propagators for constraints on those variables. Once the space is "stable", i.e. once the propagators can no longer narrow down any domains, you choose a new constraint "C" and make two spaces with the new constraints "C" and "not C".

This is due to the distributive law – $(S \wedge C) \vee (S \wedge \neg C) \equiv S$ – and so this step is also known as "distribution".

After further propagation on each cloned space, you may get a failed space or another stable one. If the latter, you continue to branch, until you get a solved space.

The rule that determines what "C" to use at each stage is called "branching strategy".
The rule that determines how you descend this "tree" looking for a solution is called the "search strategy".

Preparation

* Download and install fossil from fossil-scm.org
* Extract fossil.exe and put it in your command-line path.
* Chrome: make sure your browser is up to date (version > 15.x)
* Firefox: Install the "firebug" extension

Preparation

- Fetch the fd.js fossil repository and checkout the code in a local folder.

```
> fossil clone http://nishabdam.com:8080/fd fd.fossil
> mkdir fd
> cd fd
> fossil open ../fd.fossil
```

If you put your fossil.exe in a particular place, then qualify the “fossil” in the commands above with the path. For example, if before you execute these commands, the fossil.exe is in the current directory, then the last command must be “..\fossil open ../fd.fossil” instead.

Preparation

- Edit the tests.html file to look like this and open it in your browser.

```
<html>
    <body>
        <script src="fd.js"></script>
        <script src=”YOU.js”></script>
    </body>
</html>
```

- Create YOU.js and open in your editor.

Preparation

- Chrome: Open the Javascript console
- Firefox: Open Firebug
- Open the fd.js API documentation from http://nishabdam.com:8080/fd

We're all set!

An alternative to using the browser is to use Node.js, but its installation can be complicated and fd.js needs a bit of modification to work with Node.

A simple example

We're going to solve for `X > Y` where

```
X is in [[5,10]]
```

```
Y is in [[8,20]]
```

(Obviously, no unique solution exists.)

Figure out how the domains are supposed to change to reflect the given X > Y constraint. Can X take on the value 7? Can Y take on the value 15?

A simple example

```
 JS console:

> var S = new FD.space()
> S.decl('X', [[5,10]])
> S.decl('Y', [[8,20]])

View the variables in the space:
> ''+S
> S.vars.X.dom
> S.vars.Y.dom
```

Finite integer domains are represented by an array of intervals, where each interval is represented as a two-element array with the first element giving the lower bound of the interval and the second element giving the upper bound (inclusive).

So in this example, S.vars.X.dom[0][0] === 5 and S.vars.X.dom[0][1] === 10

A simple example

```
 JS console:

> S.gt(‘X’, ‘Y’)
> ‘’+S
```

“gt” stands for “greater than”. Other comparisons are named similarly – “lt”, “lte”, “gte”. “lte” stands for “less than or equal to”. See fd.js documentation for more information.

A simple example

```
JS console:

> S.propagate()
> ‘’+S
> S.is_solved()
```

A simple example

```
JS console:

> var S2 = S.clone();
> S2.gt('Y', S2.const(30))
> S2.propagate()
```

The “lt”, “gt” and other comparators all operate only on FD variables. The “const” method is a way to create the oxymoronic “constant variable”.

Note: The “Y > 30” constraint contradicts the current constraint store. Therefore the propagate() call should’ve thrown an exception “fail”.

A simple example

```
JS console:

> var S2 = S.clone();
> S2.eq('Y', S2.const(9))
> S2.propagate()
> ''+S2
> S2.is_solved()
```

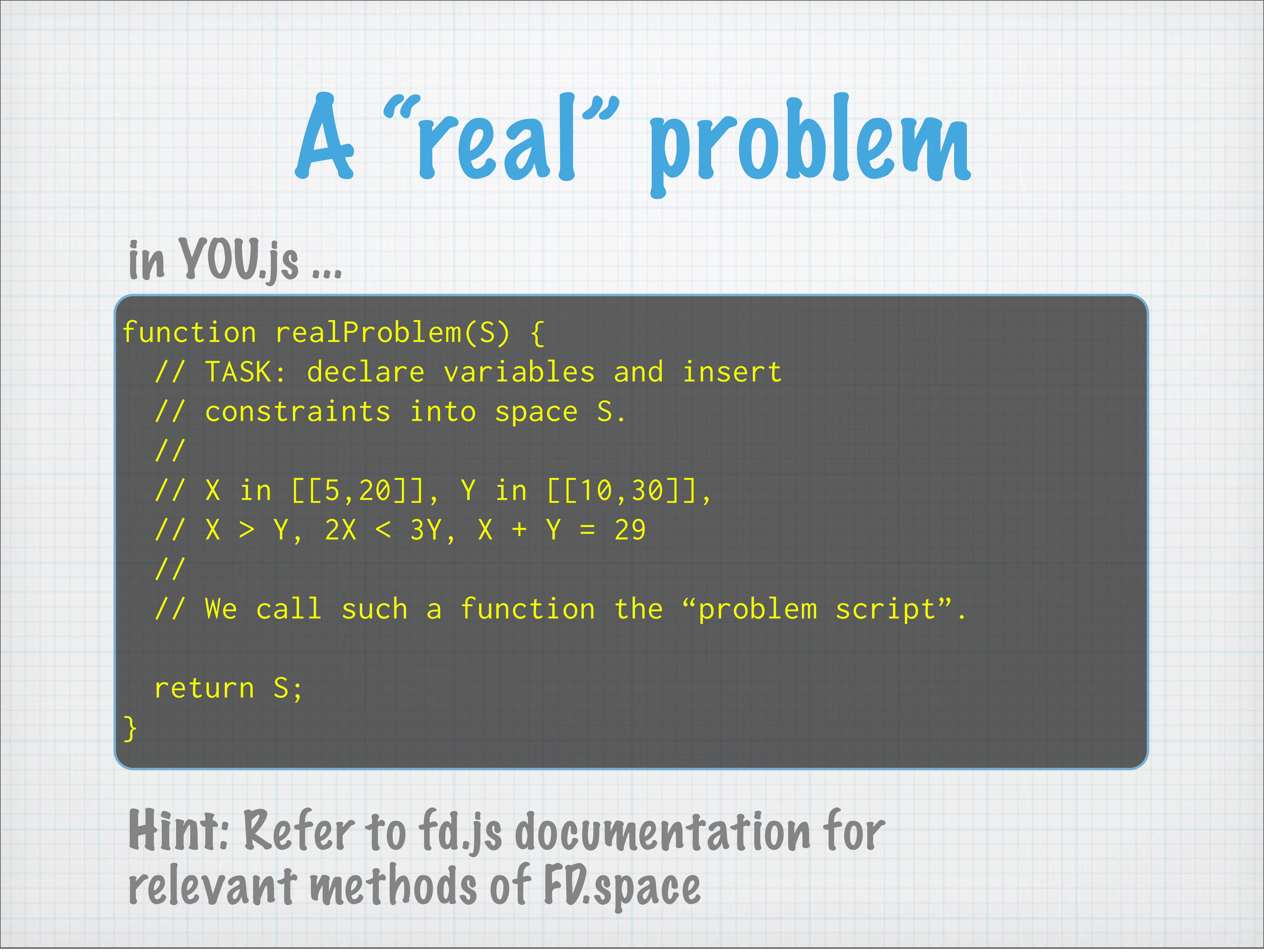
In this case, we're fixing the value of Y and that ends up determining the value of X.

A "real" problem

Solve for X and Y where

```
X > Y and 2X < 3Y and X + Y = 29

X in [[5, 20]]

Y in [[10,30]]
```

Figure out how the domains are supposed to change to reflect the given constraints. See next slide for the way you're expected to express these constraints as a "problem script".



A “real” problem

in YOU.js ...

```
function realProblem(S) {
  // TASK: declare variables and insert
  // constraints into space S.
  //
  // X in [[5,20]], Y in [[10,30]],
  // X > Y, 2X < 3Y, X + Y = 29
  //
  // We call such a function the “problem script”.

  return S;
}
```

Hint: Refer to fd.js documentation for relevant methods of FD.space

You’ll need the following methods on the space (see documentation on what they do and how to use them) – S.decl, S.const, S.plus, S.times, S.gt, S.lt

A “real” problem

in YOU.js ...

```
 function realProblem(S) {
    S.decl(‘X’, [[5,20]]);
    S.decl(‘Y’, [[10,30]]);
    S.gt(‘X’, ‘Y’);
    S.plus(‘X’, ‘Y’, S.const(29));
    S.lt(S.times(S.const(2), ‘X’), S.times(S.const(3), ‘Y’));
    return S;
}
```

try in JS console (after refresh) ...

```
> var S = realProblem(new FD.space())
> S.propagate()
> ‘’+S
```

A “real” problem

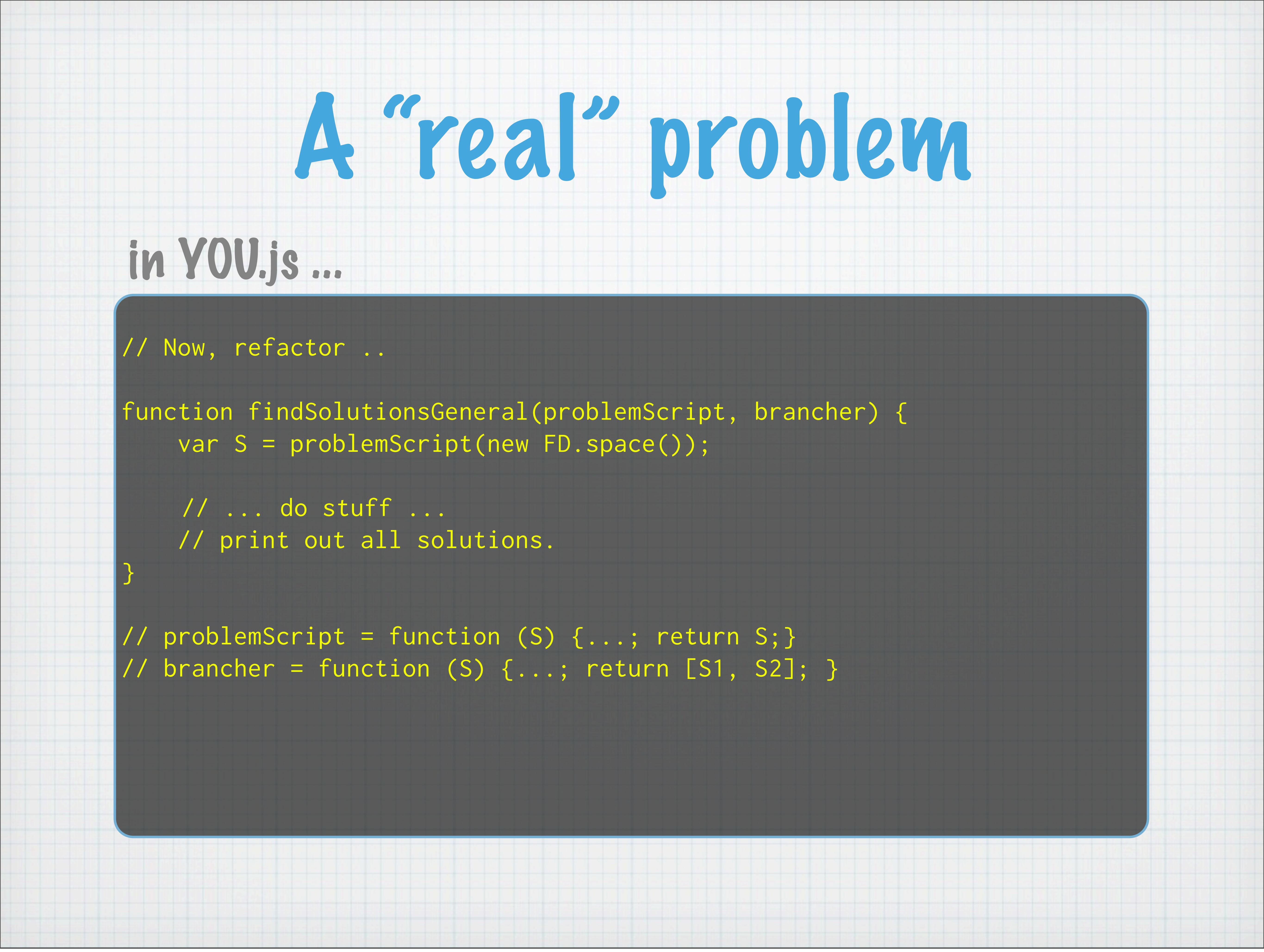
in YOU.js ...

```
function findSolutions(S, varname) {
    // Try out each allowed value of the given
    // variable, stabilize the space by running
    // propagators and print out all solutions.
    //
    // Hint: You need to S.clone() and insert
    // new constraints into the cloned space.
    //
    // Hint: Beware, calling propagate() may throw
    // “fail” as an exception.
}

var S = realProblem(new FD.space());
findSolutions(S, ‘X’);
```

Implement the “findSolutions” function as shown in this slide. The function is expected to print out all possible solutions that can be derived starting from the given space S, by trying out different values for the variable named varname.

Be aware that “varname” is a regular javascript variable whose value is expected to be a javascript string that gives the name of a finite domain variable.

You can use console.log(expr) to print out the value of the expr on the javascript console.



A “real” problem

in YOU.js ...

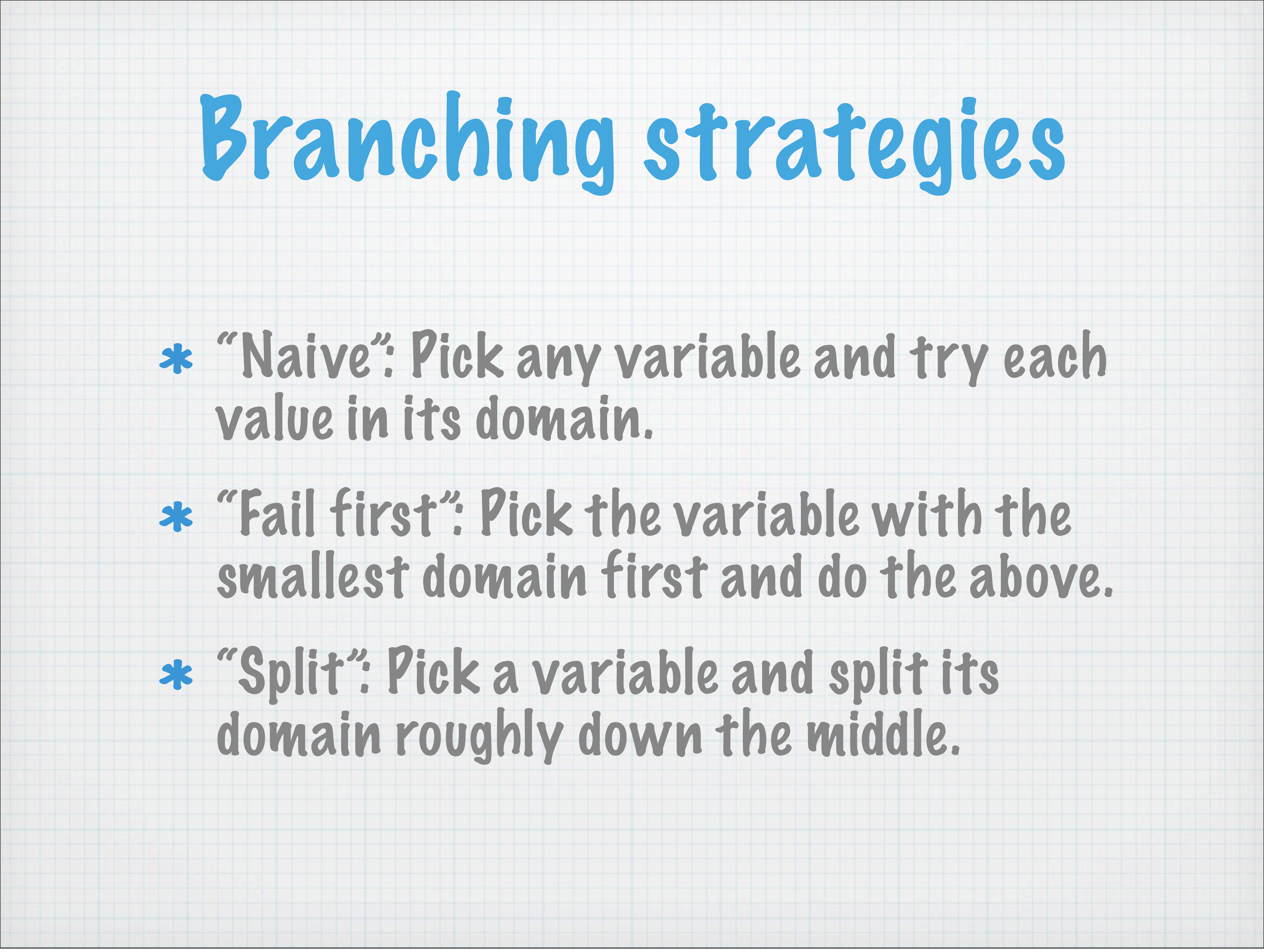
```
// Now, refactor ..

function findSolutionsGeneral(problemScript, brancher) {
    var S = problemScript(new FD.space());

     // ... do stuff ...
    // print out all solutions.
}

// problemScript = function (S) {...; return S;}
// brancher = function (S) {...; return [S1, S2]; }
```

Your previous implementation of “findSolutions” solved a specific problem. Your task is to generalize that function by abstracting over the problemScript and the way you generated spaces by cloning ... which we’ll call the brancher.

The problemScript is expected to be a function that will inject constraints into the given space and return it.

The brancher is a function that takes a space, determines a constraint C based on the current domains of the variables in the space, and returns two new cloned spaces that have the additional constraints “C” and “not C” respectively, as an array of two spaces. You are free to extend this to an array of N spaces if you prefer.

A search problem involves the specification of a set of constraints (the “problemScript”), a strategy for trying different values of variables (the “brancher”) and a strategy for finding solution spaces starting from a space (the “findSolutionsGeneral”). This function spec therefore covers all three aspects of a constraint programming approach to solving a problem.



Branching strategies

* "Naive": Pick any variable and try each value in its domain.
* "Fail first": Pick the variable with the smallest domain first and do the above.
* "Split": Pick a variable and split its domain roughly down the middle.

This slide and the next describe some strategies already implemented in the fd.js module.

Search strategies

- "Depth first": Fix the values of undetermined variables one by one until you get a solution. If you get a failure, back off and try a different value.
- "Branch and bound": If you find a solution, try to find a "better" solution on your next attempt.