Scientific Computing

Feb 21, 2025

Announcements

→ HW 3 due Wednesday, March 5 at 11:59pm

→ Wednesday, March 5 is also the in-person midterm exam

→ Friday, March 7, no lecture, extra office hours while you work on take-home (time TBD)

Today

→ Backtracking

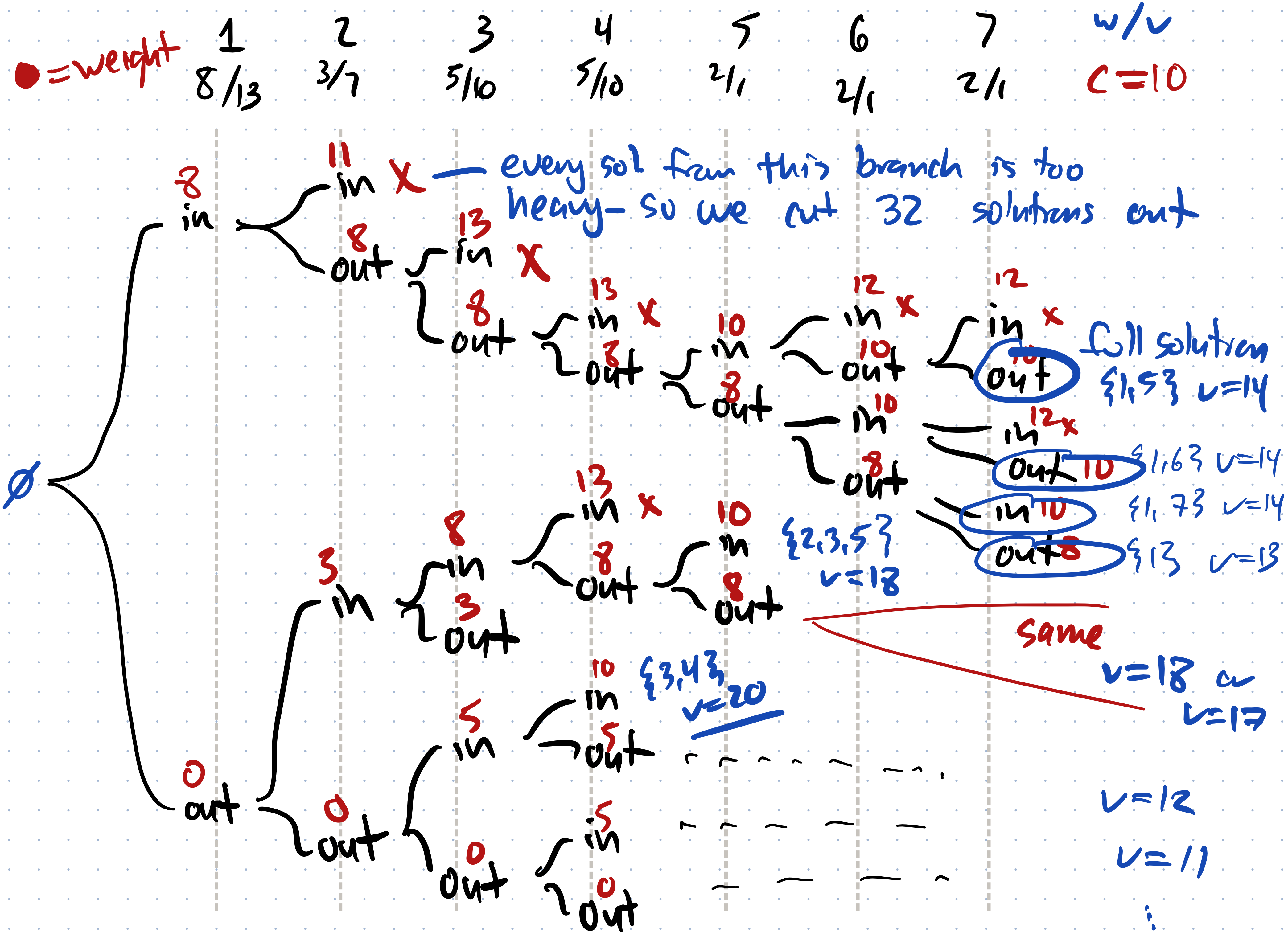
→ Branch and Bound

Office Hours:

Mon + Fri

9:30am - 10:30am

Cudahy 307



every sol from this branch is too heavy- so we cut 32 solutions out

full solution {1,5} v=14

{1,6} v=14

{1,7} v=14

{1} v=13

{2,3,5} v=18

{3,4} v=20

same

v=18 or v=17

v=12

v=11

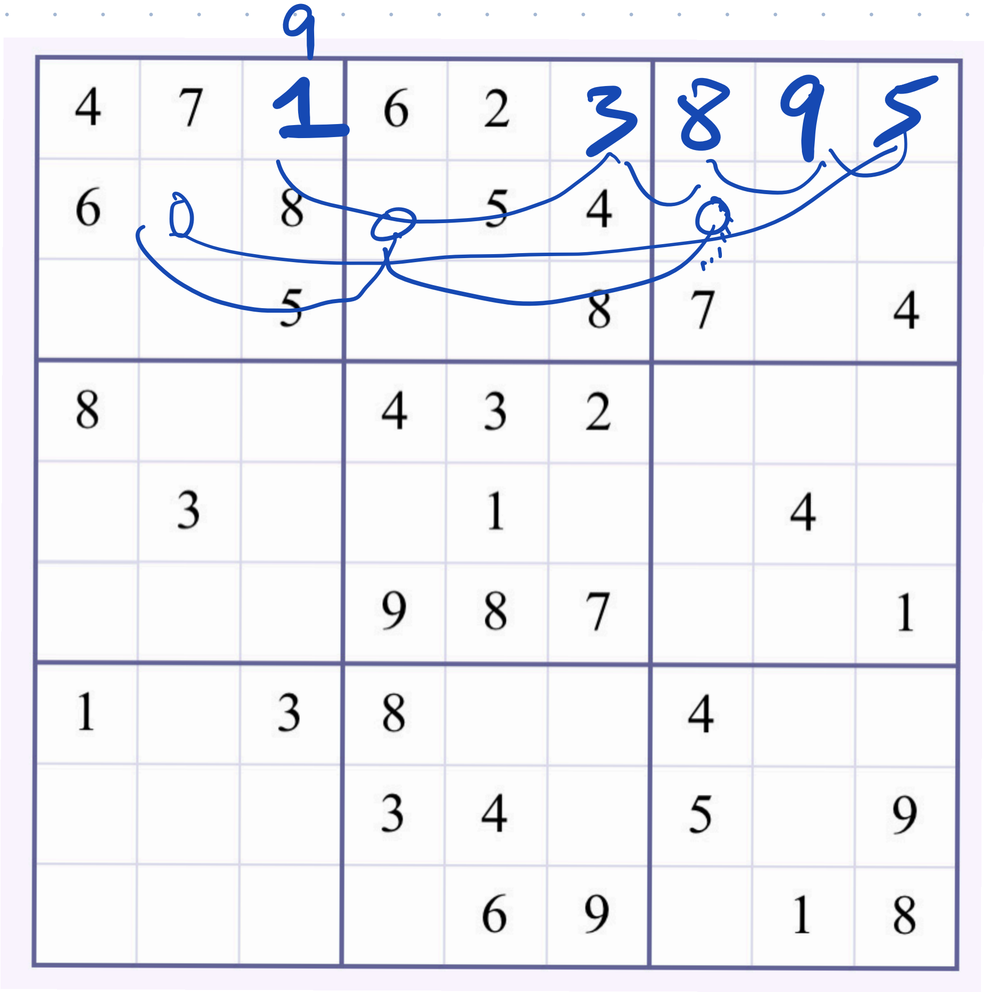
⋮

So, we are checking *or ruling out* every candidate in the search space. In bad cases (high capacity, light items), we might not rule anything out, and so in the worst case this is as bad as brute force.

[demo]

Ex #2: Sudoku

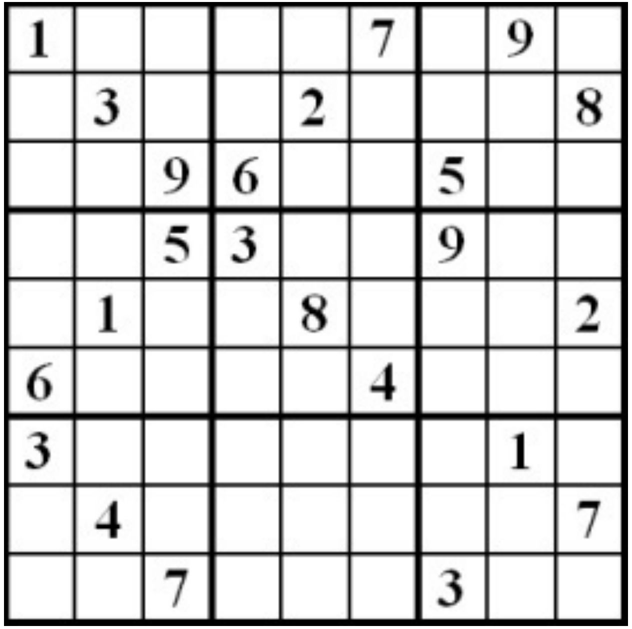
- Start filling in blank cells L-to-R then T-to-bottom.
- Start each cell at 1.
- If the cell doesn't violate a rule, move to the next cell.
- If not, bump up the value.
- If you run out of possibilities, go back to the previous cell.



4	7	9 1	6	2	3	8	9	5
6	0	8		5	4			
		5			8	7		4
8			4	3	2			
	3			1			4	
			9	8	7			1
1		3	8			4		
			3	4		5		9
				6	9		1	8

* online demo – jaypantone.com/sudoku
/sudoku-slow

"Hardest Sudoku Ever"



1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

Ex 3: Weighted Interval Scheduling

Requests $R = \{r_1, r_2, r_3, \ldots\}$

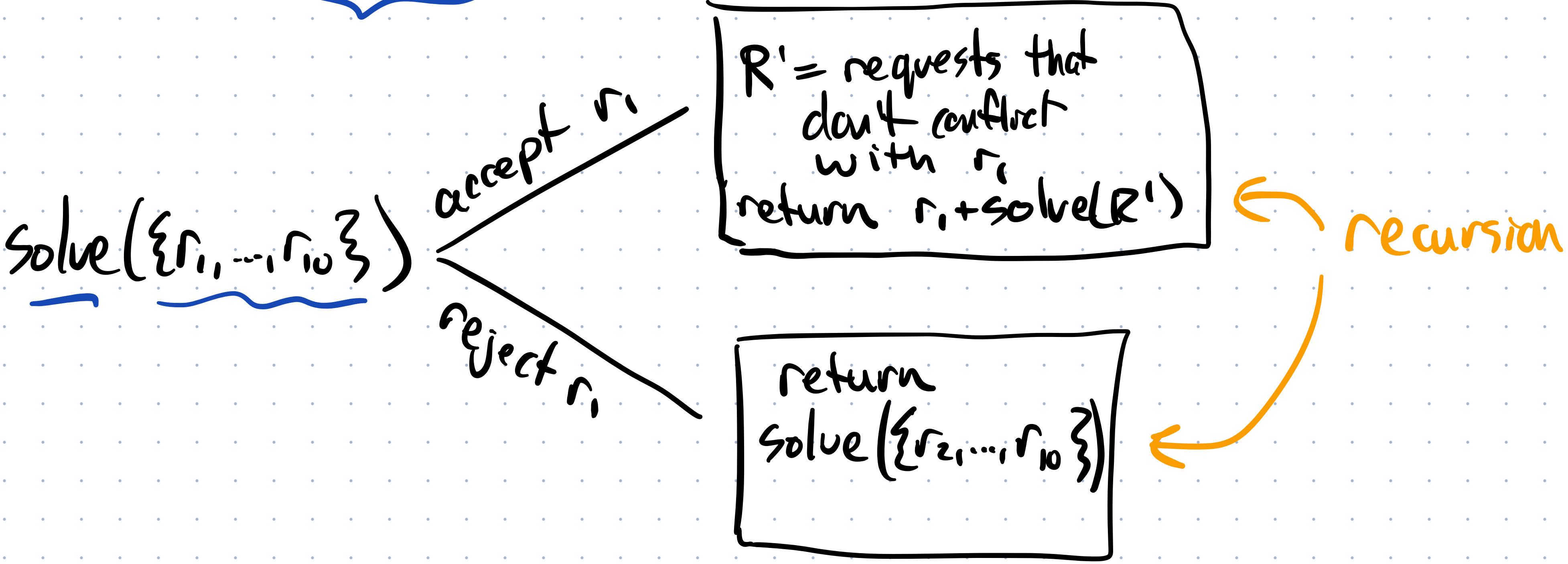
start time
end time
value

You either accept or reject each request. If you accept r_i, then in the future you can ignore requests that conflict with r_i.

This is exactly the kind of situation that recursion is perfect for because we're repeating the same logic repeatedly on subproblems.

$\underbrace{R=\{r_1,\dots,r_{10}\}}$

eliminates silly answers with meetings that conflict with r_1


solve($\{r_1,\dots,r_{10}\}$)
accept r_1
R' = requests that don't conflict with r_1
return r_1+solve(R')
reject r_1
return solve($\{r_2,\dots,r_{10}\}$)
recursion

Pseudocode

```
function solve(requests):
    # goal: return best solution that
            can be made from [requests]
    if len(requests) = 0:
        return [ ]
                                                  R'
    new_request = requests[0]
    compatible = requests compatible with new_requests
    accept_solution = [new_request] + solve(compatible)  } recursion
    reject_solution = solve(requests[1:]) — $\{r_2, r_3, \ldots r_n\}$  }
    return whichever of accept_solution and reject_solution
        has the highest value
```

[demo!]