Scientific Computing

Feb 24, 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

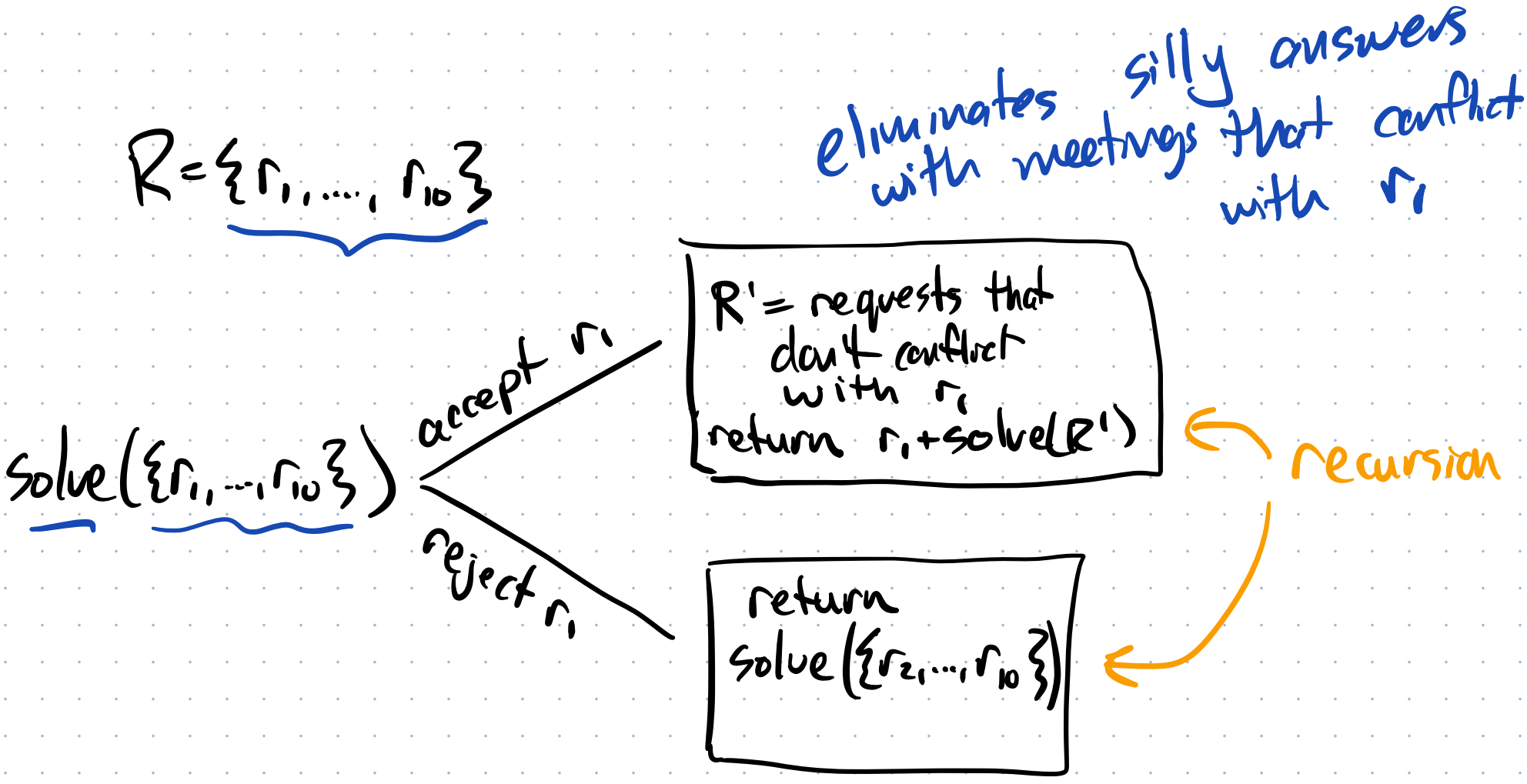
Ex 3: Weighted Interval Scheduling

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.


R = {r1, ..., r10}
eliminates silly answers with meetings that conflict with r1
Solve({r1, ..., r10})
accept r1
reject r1
R' = requests that don't conflict with r1
return r1 + solve(R')
return solve({r2, ..., r10})
recursion

Topic 8 - Branch and Bound

Recall that our problems usually have two considerations:

(1) Constraints that must be satisfied

ex: capacity of the knapsack

choosing interval requests that don't conflict

row/col/square sudoku conditions

(2) A value/score that we want to maximize or minimize among all candidates that satisfy the constraints.

Some problems are only about constraints (Sudoku, NFL scheduling).

Some problems don't really have constraints and are only about optimizing – it can depend on how you define your search space (traveling salesman)

Backtracking boiled down to:

If you build your solutions a bit at a time, you can detect early if the constraints are violated, and rule out a chunk of the search space at once.

This never considered value.

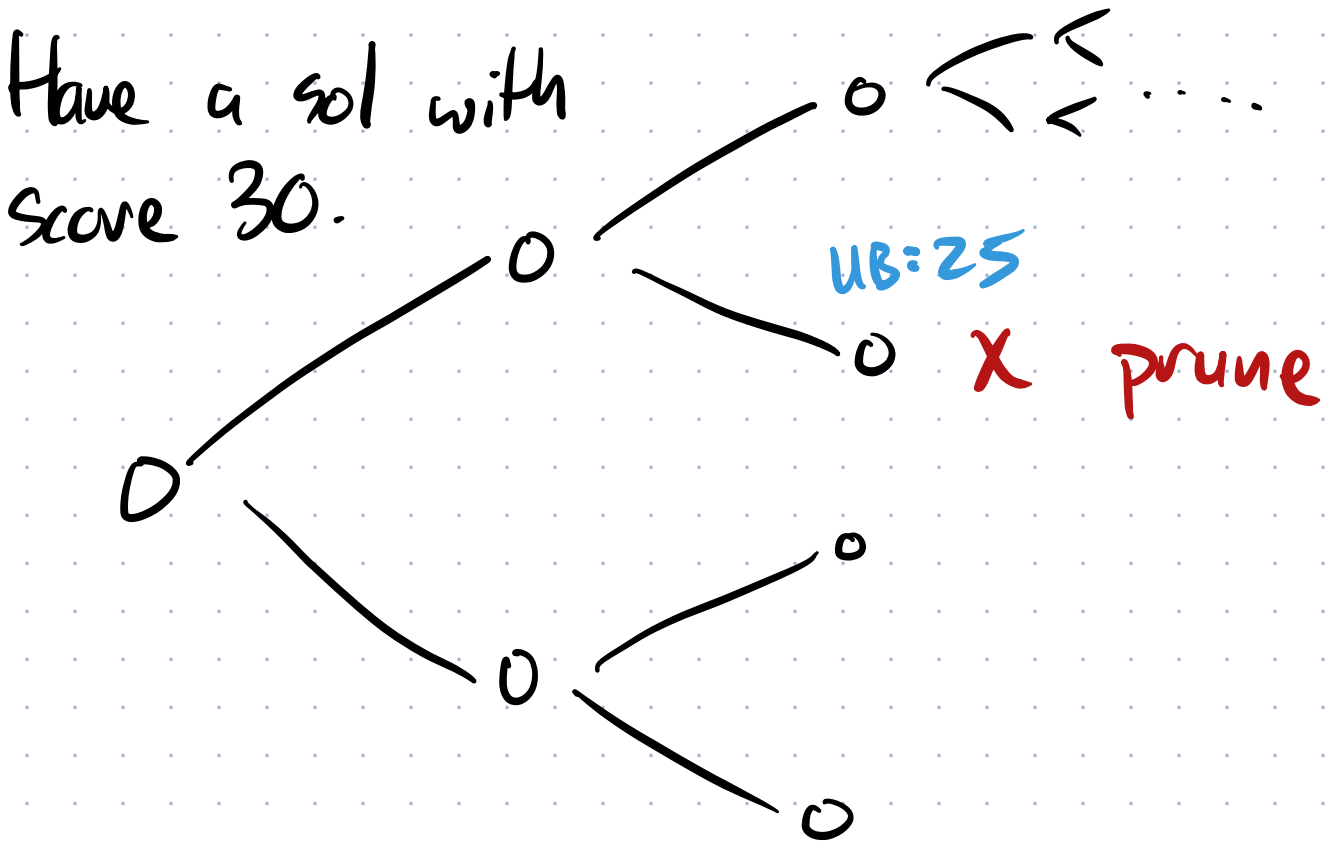
Branch and Bound is just Backtracking with an extra way to rule out a partial solution: (assuming maximization for now)

* If I've already seen a complete solution with a score of X, and there is no way to complete this partial solution in a way that beats that, prune it (stop looking at this node).

There's no way to know exactly the best you can do on completing a partial solution — if you could do that, you could just do it from the start and solve the problem right away.

Need: A way to get an upper bound on the best you could do when completing a partial solution.

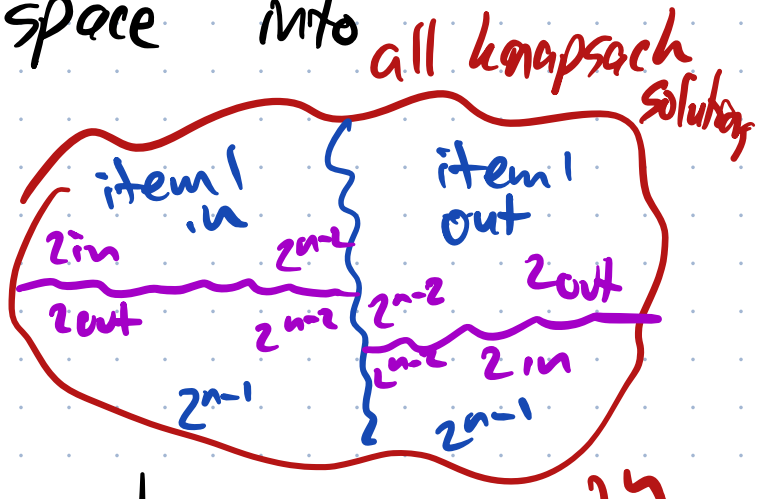
"I don't know how good I can do, but I know for sure I can't do better than Y."



We're not addressing the hard part yet: how to compute an upper bound. We'll come back to that.

Mathematical Framework for Backtracking and B+B:

(1) "making decisions to build partial solutions"
⇒ really splitting the search space into disjoint parts (subspaces)
↳ no overlap



Ex: Knapsack - Item 1 is in or out
{all subsets of items} → {subsets containing 1} and {subsets not containing 1}

Another example:

{subsets containing 1}

↘

{subsets containing 1 and 2} and {subsets containing 1 and not containing 2}

This is called branching. It doesn't have to always be into two parts.

(2) For any subspace S we create with branching, we need to be able to compute bound(S), some upper bound on the best score possible for any candidate in S.

Notes:

* We're phrasing this all for maximization.
* bound(S) has to be an upper bound. Lower bounds are easy (greedy) but useless.

Ex: Problem #5: Job Assignment

You have n tasks that need to be done and n workers. Each task has a different cost to complete depending on which worker does it. Goal: Minimize total cost.

workers \ tasks	1	2	3	4
A	3	5	(2)	2
B	(6)	8	10	8
C	2	6	4	(9)
D	10	(4)	7	5

cost: $6+4+2+9=21$

Many applications!

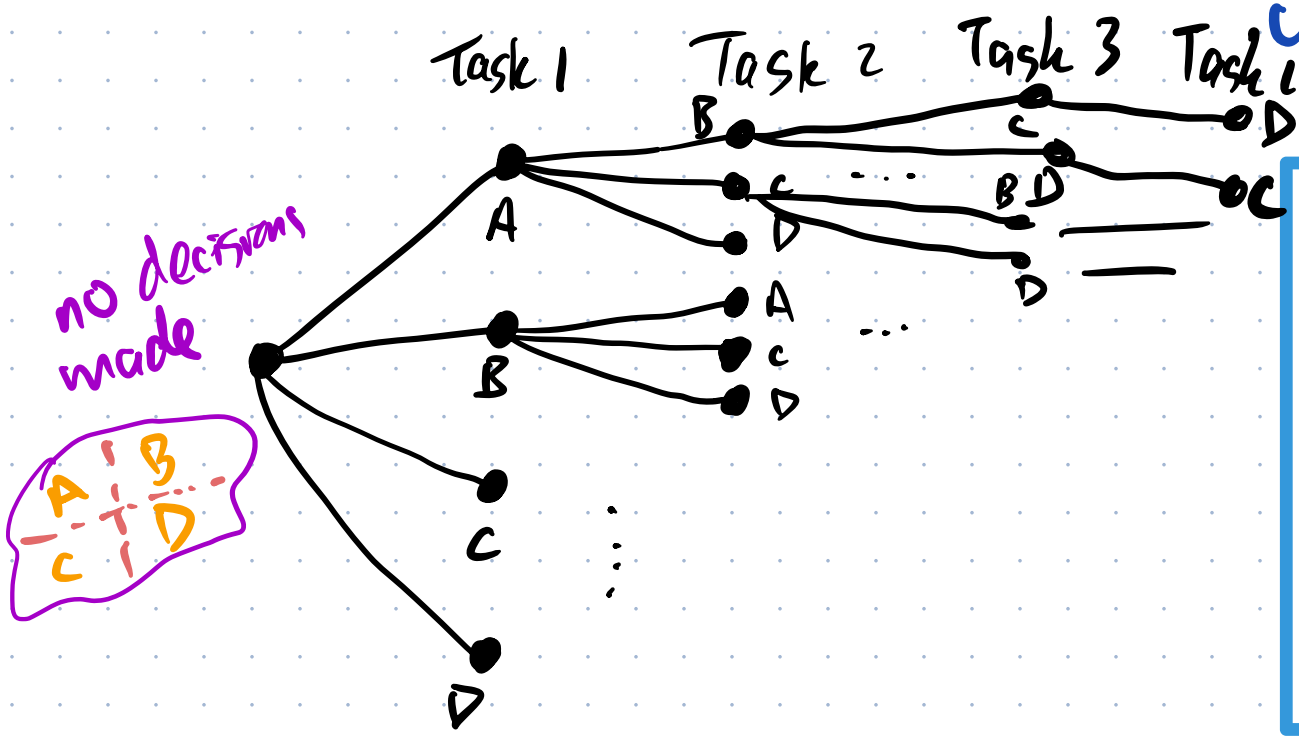
- Drivers picking up passengers
- Shipments from mines to factories

* Search Space: All assignments of workers to tasks.
  How big? $n!$

* No constraints, so backtracking alone is useless.
  (equiv. to brute force)

Branching – Pick which worker does a certain task

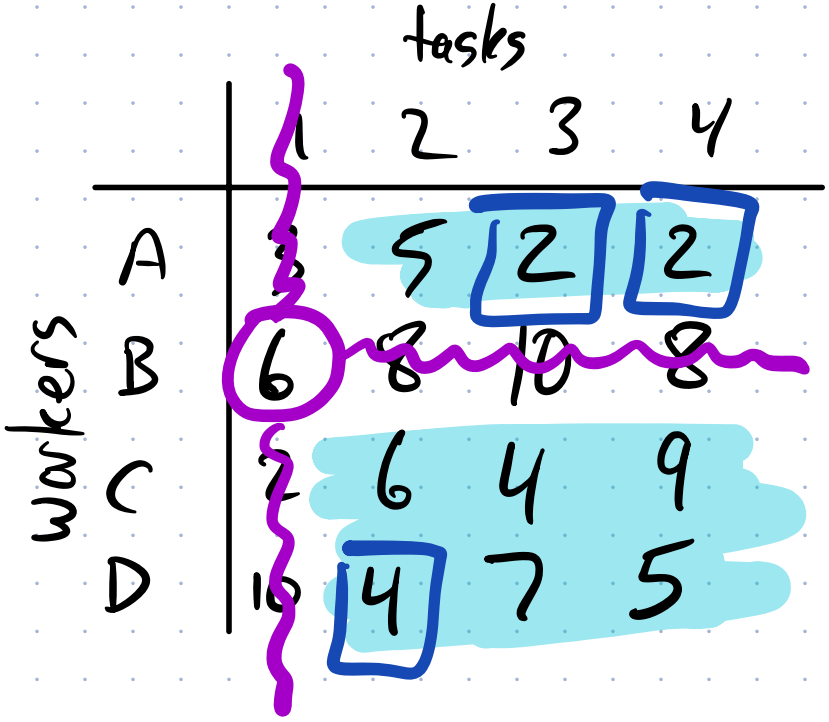
$4 \cdot 3 \cdot 2 \cdot 1 = 4!$



workers \ tasks	1	2	3	4
A	3	5	2	2
B	6	8	10	8
C	2	6	4	9
D	10	4	7	5

Bounding:

* Suppose you've already picked worker B to do Task 1.
* What is a lower bound on the best you can do to finish? (Has to be easier than actually solving the whole problem.)



One possibility: remaining

Every task will cost at least the minimum number in its column.

"We don't know what the best score is, but it definitely can't be less than X" (lower bound)

Finishing this partially built sol will cost ≥ 8. 8 is a lower bound

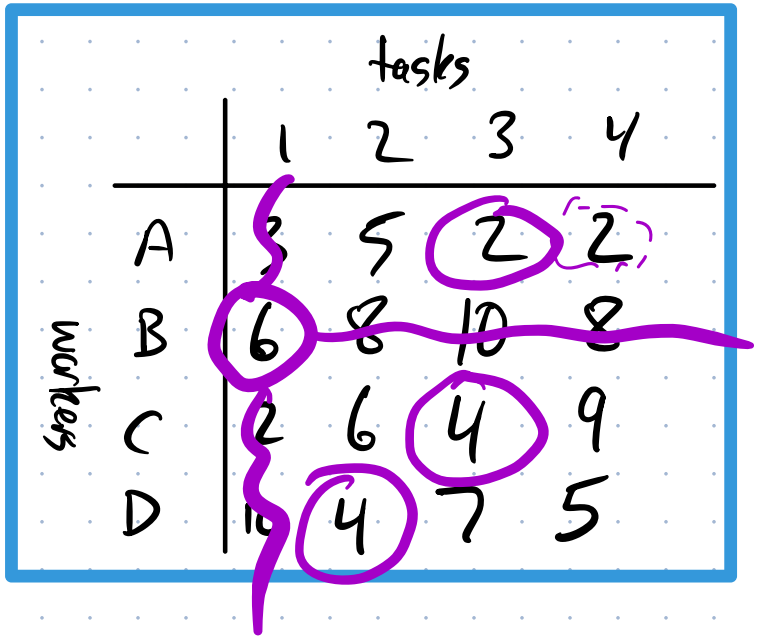
One possibility: every task will cost <u>at least</u> its minimum remaining

$4+2+2=8$

Another possibility: every worker will incur a cost <u>at least</u> the minimum of the tasks remaining.

So, finishing will cost at least <u>10</u>.

Which bound is better?



So, our lower bound will be
max(sum of smallest cost in each remaining row,
sum of smallest cost in each remaining col)
+ existing cost of selections.