Scientific Computing

Feb 26, 2025

Announcements

→ HW 2 is graded, see feedback on D2L
"***" means you can revise that problem for half-credit back, due tomorrow night

→ Solutions sent on Friday

→ 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

→ Branch and Bound

Office Hours:

Mon + Fri

9:30am - 10:30am

Cudahy 307

Homework 3 Discussion

Q1 Two Knapsack Problem

Regular Knapsack: Input: a list of items (w, v) (weight, value), capacity integer

Output: the subset of items whose total weight is $\leq$ capacity and whose total value is maximized.

To loop over the search space, loop over all subsets

$\{1,2,3\}$

$\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}$
$\{1,2,3\}$

Two Knapsack: Input: a list of items
two capacities

Homework 3 Discussion

Q1 Two Knapsack Problem

Two Knapsack: Input: a list of items
two capacities

Output: Two disjoint subsets of items such that total weight of items in K1 is $\leq C1$, and total weight of items in K2 is $\leq C2$ and total value of all items is maximized.

Items: $\{1,2,3,4,5\}$

One element of the search space is:

$\{2\}, \{1,3,5\}$

Items: {1,2,3} n

27 elements in search space

K1 / K2

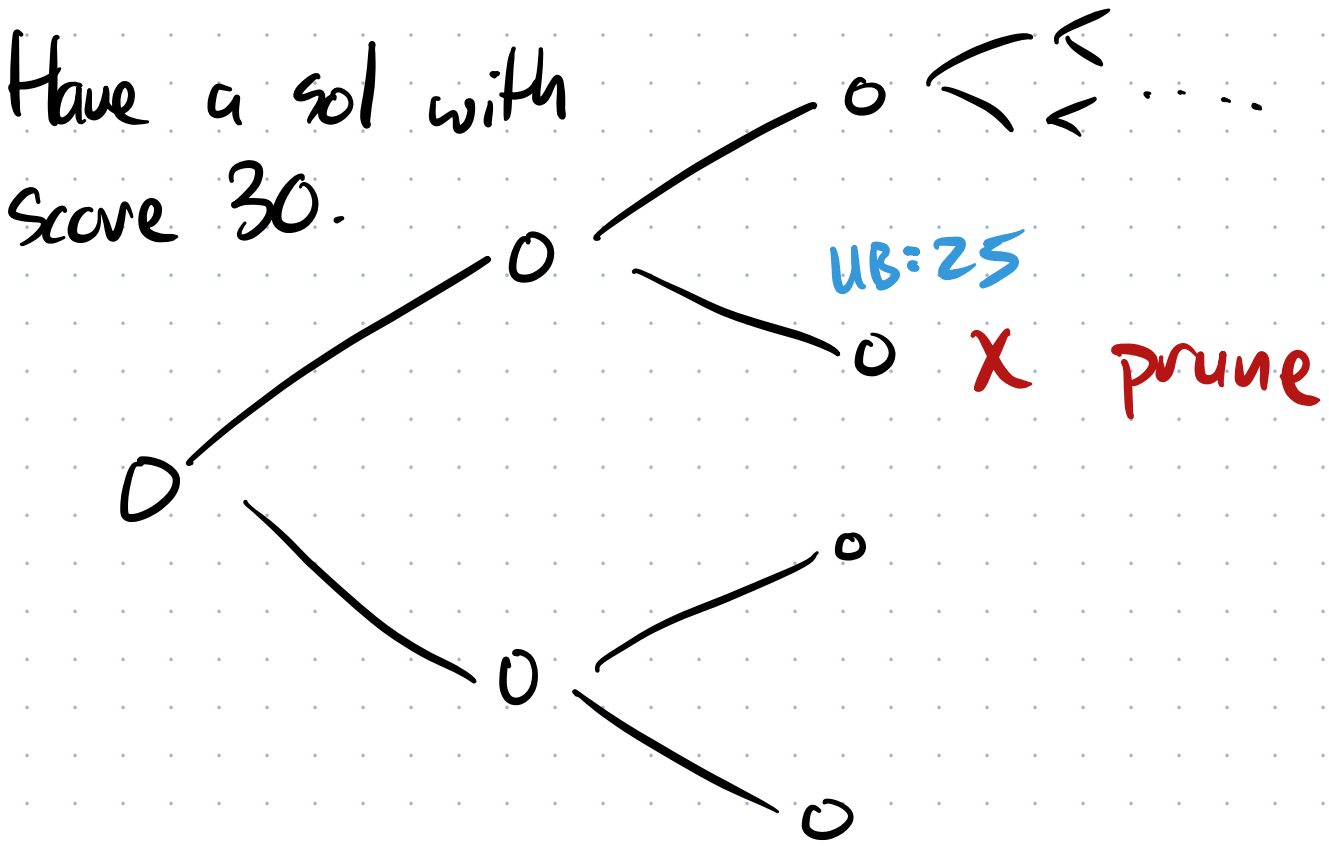
-/- (L0: 1,2,3)

1/-
-/1
2/-
-/2
3/-
-/3

1/2
2/1
1/3
3/1
2/3
3/2
12/-
13/-
23/-

-/12
-/23
-/13
123/-
12/3
13/2
23/1
1/23
2/13
3/12
-/123

n=10 1347 / 29



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

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.

		tasks			
		1	2	3	4
workers	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!

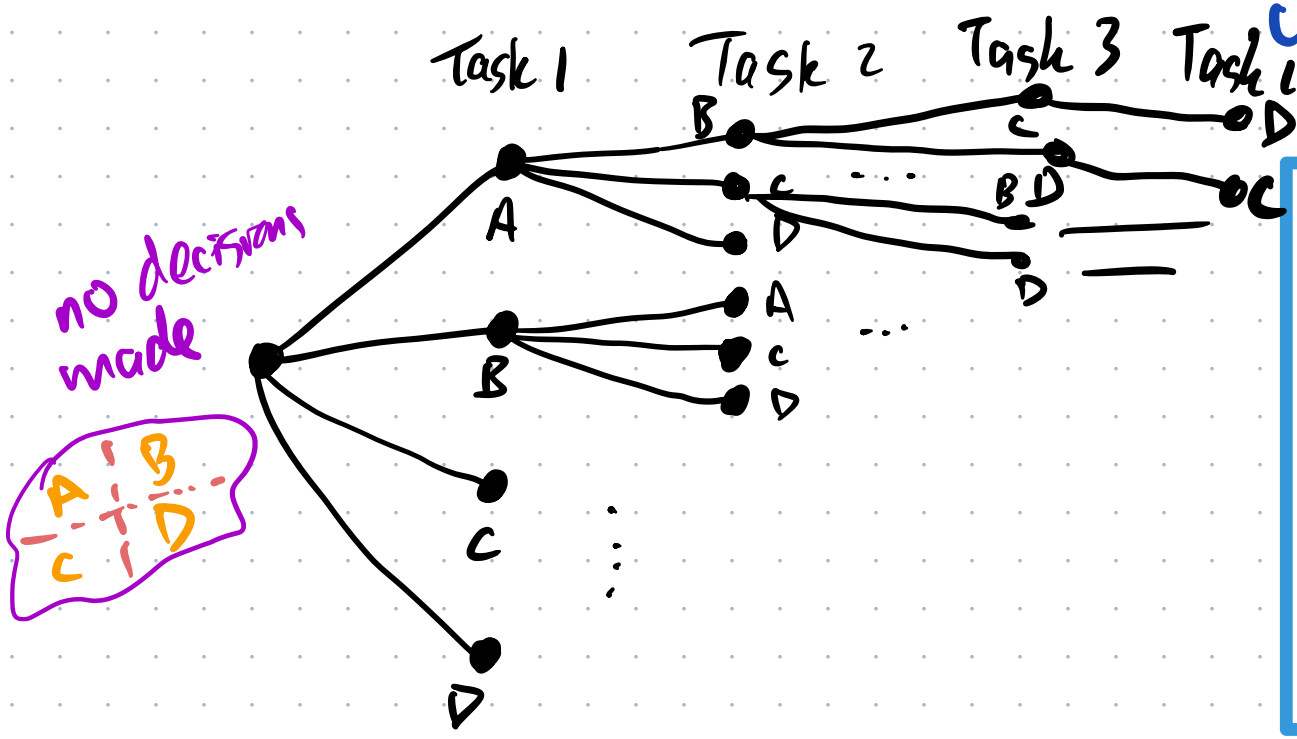
- Drivers picking up passengers
- Shipments from mines to factories

* 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

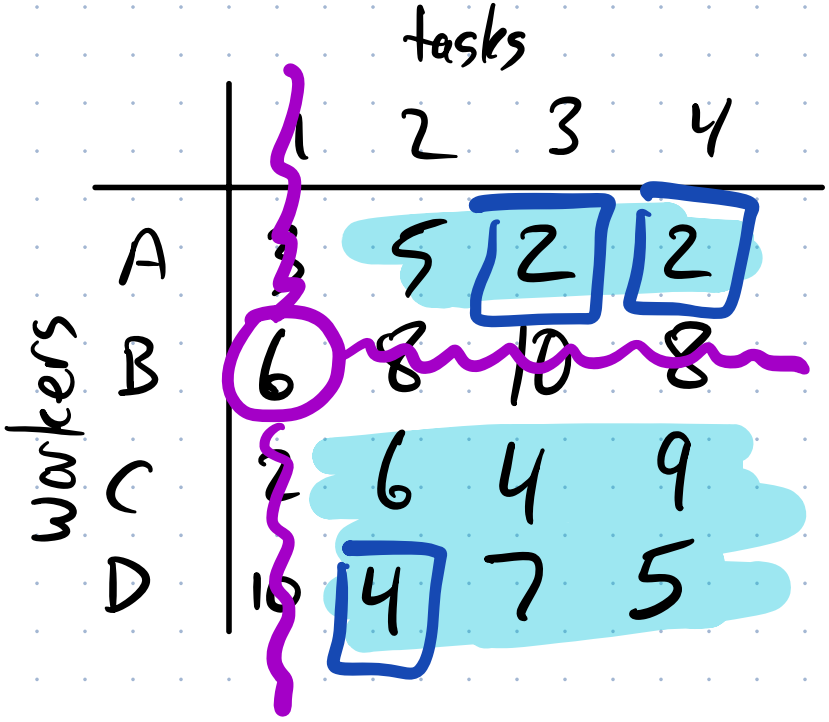
$4 \cdot 3 \cdot 2 \cdot 1 = 4!$



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.

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

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

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

$4+2+2=8$

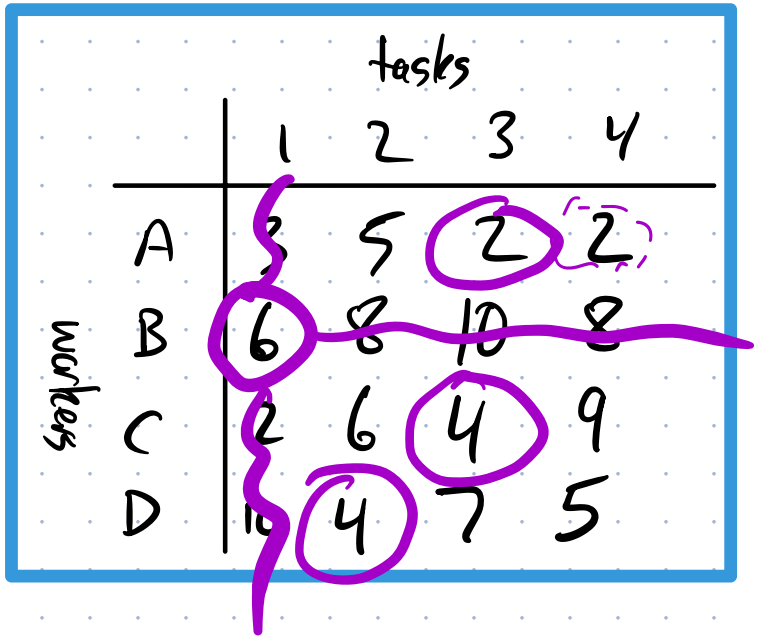
↳ Every solution down the branch has a score ≥ 14

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? Every sol down the branch has a score ≥ 16



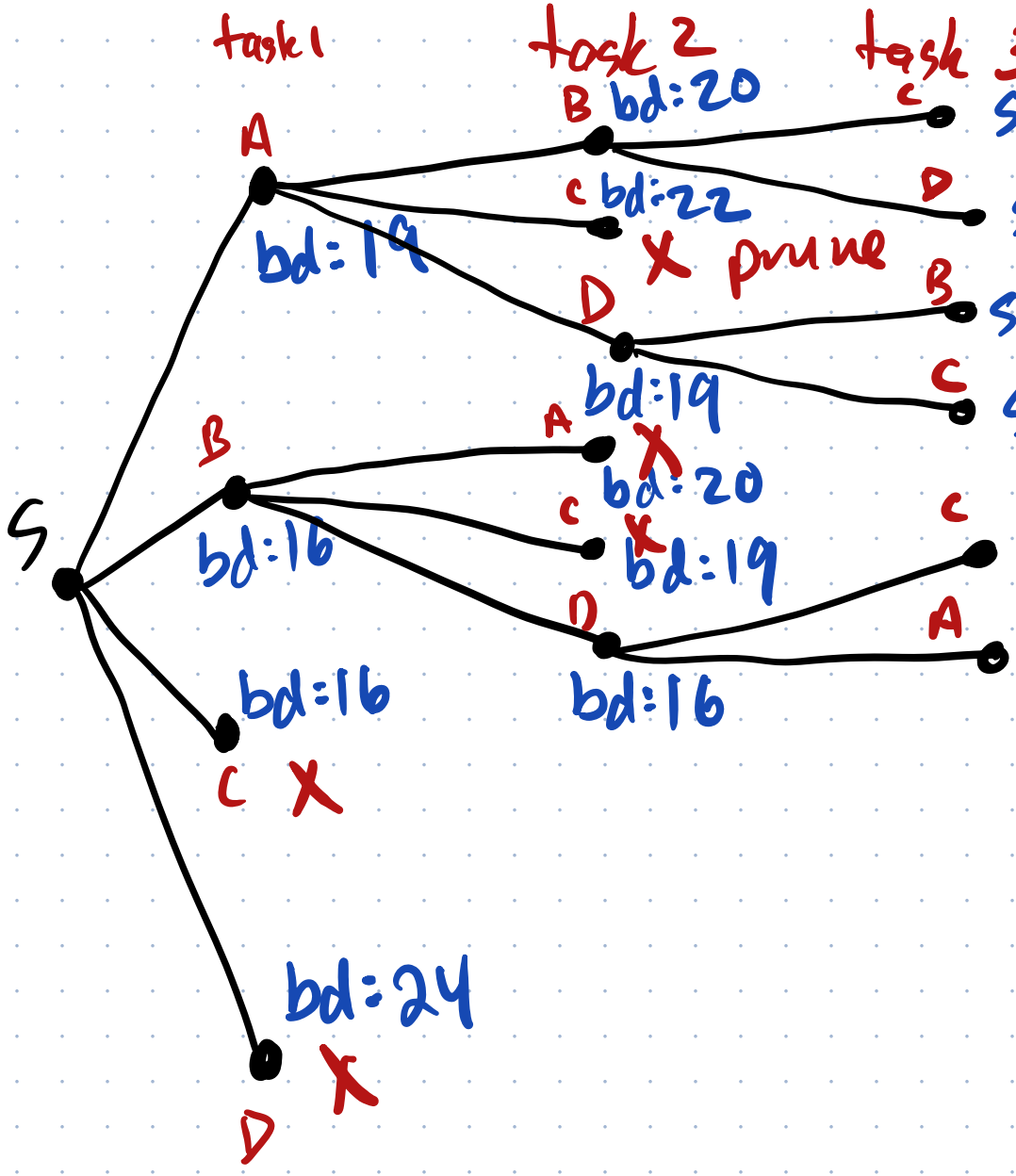
We're minimizing, so we need a lower bound to prune.

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.

$[\max(10, 8)] + 6 = 16$

Let's work through it.

best sol: ~~∞~~ ~~20~~ ~~19~~ 16



Optimal Solution

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

Notes:

* Again, the hardest part is finding a good bound! The stronger, the better.

* At the start, we didn't have a best_sol to do any pruning until we branched down to a single candidate. If we found a candidate before we started B+B, maybe there would have been more pruning.
  - Pick a few random solutions.
  - Pick a greedy solution.

Can get a greedy solution with score 16!

That would be proved optimal with very little branching.

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

General Procedure:

search space

```
function bb(S, best_sol = None):
    if best_sol is None:
        best_score = -∞
    else:
        best_score = score(best_sol)
    if |S| = 1:                          (only has a single complete candidate)
        candidate = the one thing in S
        value = score(candidate)
        if value > best_score:
            return candidate
        else:
            return best_sol
```

base case

[In abstract discussions, we've maximizing]

General Procedure:

```
function bb( , best_sol = None):
    if best_sol is None:
        best_score = -∞
    else:
        best_score = score(best_sol)
    if |S| = 1:
        candidate = the one thing in S
        value = score(candidate)
        if value > best_score:
            return candidate
        else:
            return best_sol
```

else

branch into two, but could be 3, 4, etc.

```
S1, S2 = branch(S)
if bound(S1) > best_score:
    best_sol = bb(S1, best_sol)
    best_score = score(best_sol)
if bound(S2) > best_score
    best_sol = bb(S2, best_sol)
    best_score = score(best_sol)
return best_sol
```