Branch and Bound

CSX3009 Algorithm Design
Kwankamol Nongpong, Ph.D.
Department of Computer Science
Assumption University

If brute force checks all solutions, how can we safely avoid checking some?

Where is the waste in bruteforce?

Knapsack Problem

- A relief organization must pack supplies into a rescue helicopter with limited weight.
- Each item has different medical priority.
- How do we guarantee the best selection?

Problem Formulation

- A collection of items
  - Weight
  - Value
- Capacity

- Objective: To maximize the value

Bruteforce

- Try everything (all subsets)
- Time complexity: $O(2^n)$

Bruteforce works but explodes very fast.

Pruning

- A general technique to eliminate unnecessary branches in a search tree.
- It is like cutting off branches of a tree that are dead or don't lead anywhere productive.

Backtracking

- Stop early when reaching invalid paths.
- If the weight exceeds the capacity, stop the branch.

Backtracking prunes invalid paths.

Greedy

- Fast but risky
- Sort the item by value/weight ratio.

- Why does the local optimization fail?

Greedy sacrifices correctness for speed.

Branch and Bound

- A systematic algorithm for solving combinatorial optimization problems, which employs pruning.
- It consists of 3 components:
  - Branching: Explore nodes (whether to include or exclude)
  - Bounding: The best possible value (estimate)
  - Pruning:
    - If upper bound < current_best (for maximization problem)
    - If lower bound > current_best (for minimization problem)

Branching

- Divides the overall problem into smaller, more manageable subproblems.
- This will form a tree-like structure.
- What does each node represent?

Bounding

- Calculates lower or upper bounds for the optimal solution within each subproblem.
- This helps to determine the best possible solution that can be found in that part of the search space.

Properties of Strong Bound

Property	Reason
Tight	Prune more branches
Admissible	Preserve correctness
Cheap to Compute	Bound cost should not outweigh pruning gains
Monotonic	Improve pruning consistency

Once your restrict choices, your best-case outcome should not improve.

Monotonic Bound: Knapsack Problem

- Upper bound is computed using fractional knapsack.
- As we pick more items, the remaining capacity decreases.
  - The maximum possible value (in the future) should stay the same or decrease.
  - It should never go up.

Example

- Capacity = 10

Item	Value	Weight
A	40	2
B	50	3
C	100	5
D	95	5

Step 1: Calculate the Ratio

- Capacity = 10

Item	Value	Weight	Ratio
A	40	2	
B	50	3	
C	100	5	
D	95	5	

Step 2: Calculate the Bound

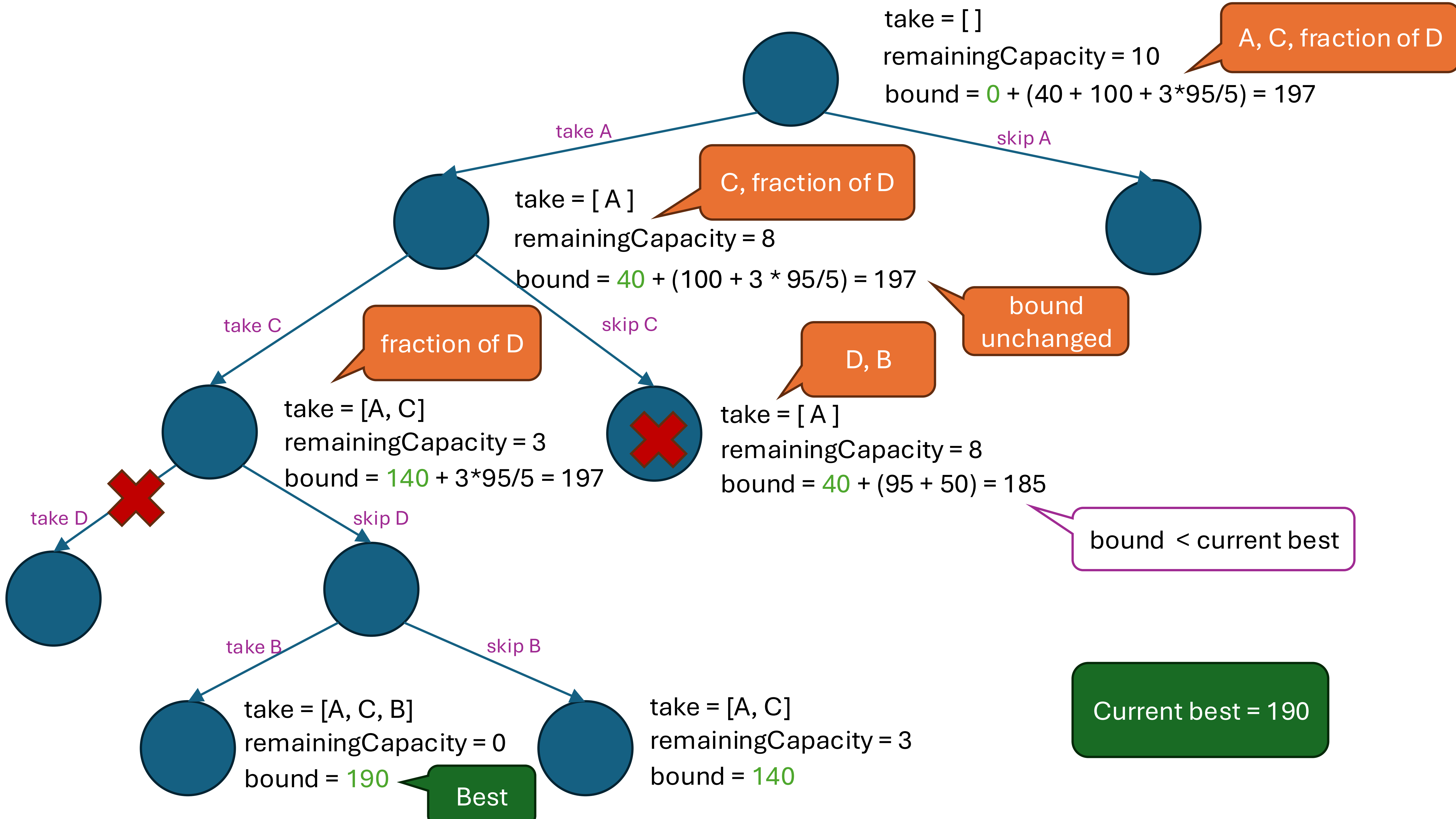
- Sort items by the value-to-weight ratio.

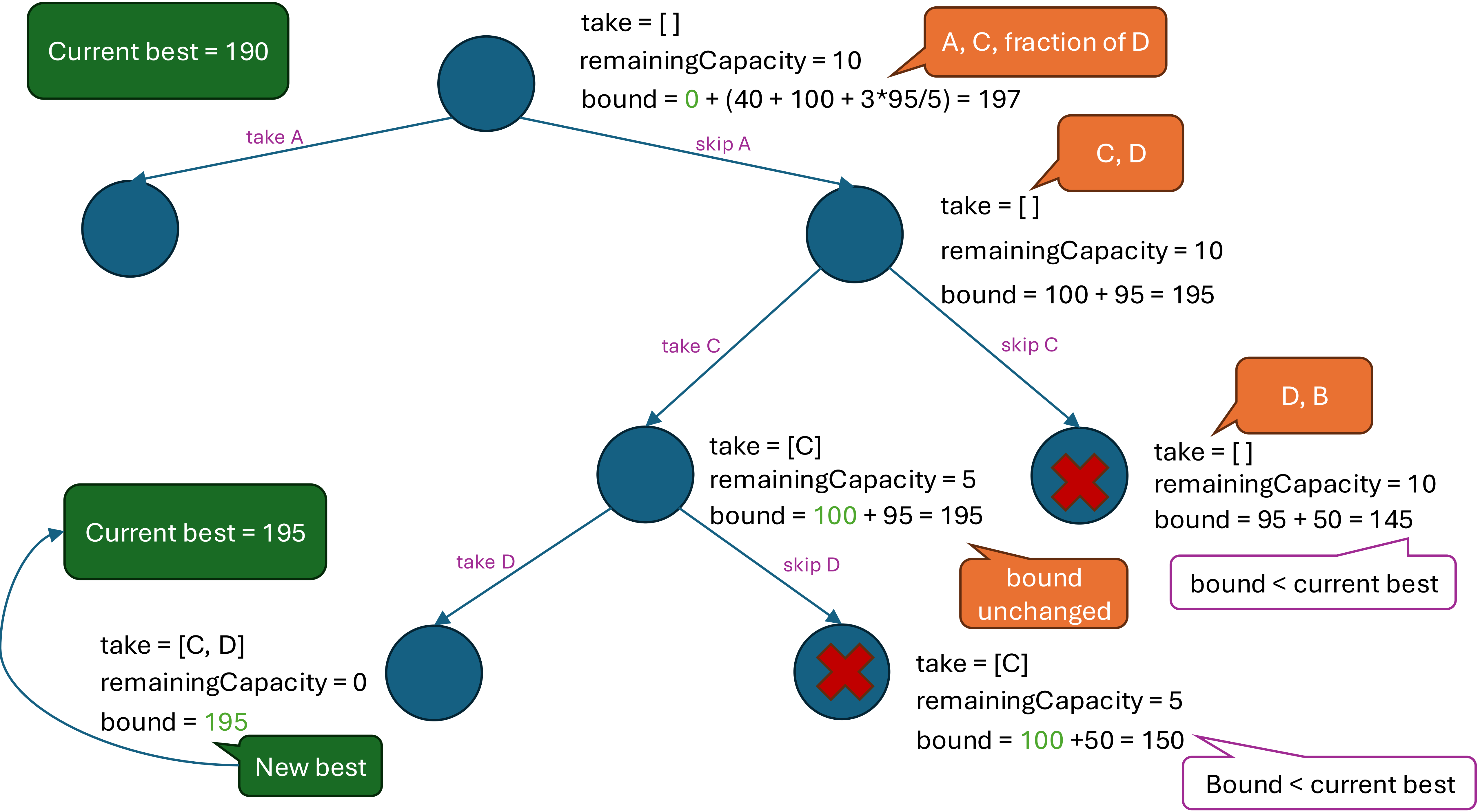
Item	Value	Weight	Ratio
A	40	2	20
C	100	5	20
D	95	5	19
B	50	3	16.6

- Calculate the bound

$$UpperBound = CurrentValue + RemainingCapacity * NextItemRatio$$

If the next item doesn't fit, fill the rest with a fraction of it.


take = [ ]
remainingCapacity = 10
bound = 0 + (40 + 100 + 3*95/5) = 197
A, C, fraction of D
take A
skip A
take = [A]
remainingCapacity = 8
bound = 40 + (100 + 3 * 95/5) = 197
C, fraction of D
bound unchanged
take C
skip C
fraction of D
take = [A, C]
remainingCapacity = 3
bound = 140 + 3*95/5 = 197
D, B
take = [A]
remainingCapacity = 8
bound = 40 + (95 + 50) = 185
bound < current best
take D
skip D
take B
skip B
take = [A, C, B]
remainingCapacity = 0
bound = 190
Best
take = [A, C]
remainingCapacity = 3
bound = 140
Current best = 190


Current best = 190
take = [ ]
remainingCapacity = 10
bound = 0 + (40 + 100 + 3*95/5) = 197
A, C, fraction of D
take A
skip A
C, D
take = [ ]
remainingCapacity = 10
bound = 100 + 95 = 195
take C
skip C
D, B
take = [C]
remainingCapacity = 5
bound = 100 + 95 = 195
take = [ ]
remainingCapacity = 10
bound = 95 + 50 = 145
Current best = 195
take D
skip D
bound unchanged
bound < current best
take = [C, D]
remainingCapacity = 0
bound = 195
take = [C]
remainingCapacity = 5
bound = 100 +50 = 150
New best
Bound < current best

Pruning

- Uses upper (or lower) bound to eliminate subproblems that cannot possibly contain the global optimum.
  - comparing the estimated solution of a subproblem against the best feasible solution found so far
- If such a bound is worse than the current best-known solution, the node is pruned.
  - i.e., not explored further

What about dynamic programming?

Discussion

- When is Branch & Bound better than DP?
- When is DP better than Branch & Bound?
- What makes a good bound?

Key Takeaway

Branch & Bound

- Best when state space is too large.
- Tight upper bound can eliminate most branches.
- Want early solutions without the full table computation.

The runtime depends on the heuristic.

Dynamic Programming

- Many overlapping subproblems
- State space is bounded
- Branch & Bound can’t prune much.
- Deterministic runtime

DP is best when there are a lot of **recomputations**.

The enemy is not the search size, but the recompution.

Branch and Bound is best when we can avoid exploring most of the search space.

What Makes a Good Bound?

- Can be computed quickly.
- Estimate the best possible solution.
- The tighter the bound, the more branches that can be safely pruned.
  - without missing the optimal solution.

A good bound is optimistic, tight and cheap.