Technische
Universität
Braunschweig

Institut für Betriebssysteme
und Rechnerverbund
Algorithmik

Algorithm Engineering
Lecture 5: Optimization Fundamentals

Transition to a Different Type of Optimization

Transition to a Different Type of Optimization

First half: Optimize running time/CPU usage/…

Now: Solve optimization problems (different definitions of *solve*)

This lecture: Overview of techniques, applied to Knapsack

Knapsack

Knapsack

Knapsack

Knapsack problem

Given:

Knapsack

Knapsack problem

Given:

- n items x_i with weight w_i and price p_i,

Knapsack

Knapsack problem

Given:

- n items x_i with weight w_i and price p_i,
- capacity limit c.

Knapsack

Knapsack problem

Given:

- n items x_i with weight w_i and price p_i,
- capacity limit c.

Goal: select which x_i to take.

maximize $\sum_{x_i \text{ taken}} p_i$ s.t.

Knapsack

Knapsack problem

Given:

- n items x_i with weight w_i and price p_i,
- capacity limit c.

Goal: select which x_i to take.

maximize $\sum_{x_i \text{ taken}} p_i$ s.t.

$$\sum_{x_i \text{ taken}} w_i \leq c.$$

Approaches

Different approaches:

- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Dynamic Programming

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:

  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:

  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1) \quad p_i + K(w - w_i, i-1) \}$$

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0					
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0				
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0			
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5		
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0						
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0					
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0				
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4			
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5		
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0						
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0					
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3				
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4			
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5		
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0						

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2					

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3				

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5			

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6		

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Dynamic Programming

- Solve large instance by solving subproblems and extending solutions
- Depending on what book you read: consider subproblems in structured order
- For Knapsack: assume that w_i are integer
- One subproblem $K(w, i)$ for each $(w, i) : 0 \leq w \leq c,\ 0 \leq i \leq n$:
  Optimal price with total weight $\leq w$ and items $x_1, \ldots, x_i$?

$$K(w,i) = \max\{ K(w, i-1),\ p_i + K(w - w_i, i-1) \}$$

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Dynamic Programming

Dynamic Programming

- Occurs a lot in theoretical works:

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms
  - Often, the table sizes are far from practical feasibility

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms
  - Often, the table sizes are far from practical feasibility
- Fairly rarely the best choice in practice

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms
  - Often, the table sizes are far from practical feasibility
- Fairly rarely the best choice in practice
- Typically mostly for efficiently solvable problems

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms
  - Often, the table sizes are far from practical feasibility
- Fairly rarely the best choice in practice
- Typically mostly for efficiently solvable problems
- Know if your problem is efficiently solvable; Knapsack is the exception, not the rule!

Dynamic Programming

- Occurs a lot in theoretical works:
  - Approximation algorithms/approximation schemes
  - Ad-hoc exact algorithms (e.g., for Knapsack)
  - FPT algorithms
  - Often, the table sizes are far from practical feasibility
- Fairly rarely the best choice in practice
- Typically mostly for efficiently solvable problems
- Know if your problem is efficiently solvable; Knapsack is the exception, not the rule!
- Sometimes a ‘recursive’ algorithm (memoization) may be better than a strict table!

Approaches

Different approaches:

- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Greedy Knapsack Algorithm

Greedy Knapsack Algorithm

Idea: sort items by non-increasing price per weight (efficiency $e_i = \frac{p_i}{w_i}$)

Greedy Knapsack Algorithm

Idea: sort items by non-increasing price per weight (efficiency $e_i = \frac{p_i}{w_i}$)

In that order, take them if they still fit.

Greedy Knapsack Algorithm

Idea: sort items by non-increasing price per weight (efficiency $e_i = \frac{p_i}{w_i}$)

In that order, take them if they still fit.

Result: fast $O(n \log n)$ valid (heuristic) solution, may be suboptimal

Greedy Knapsack Algorithm

Idea: sort items by non-increasing price per weight (efficiency $e_i = \frac{p_i}{w_i}$)

In that order, take them if they still fit.

Result: fast $O(n \log n)$ valid (heuristic) solution, may be suboptimal

In terms of bounds: this gives a *lower* bound!

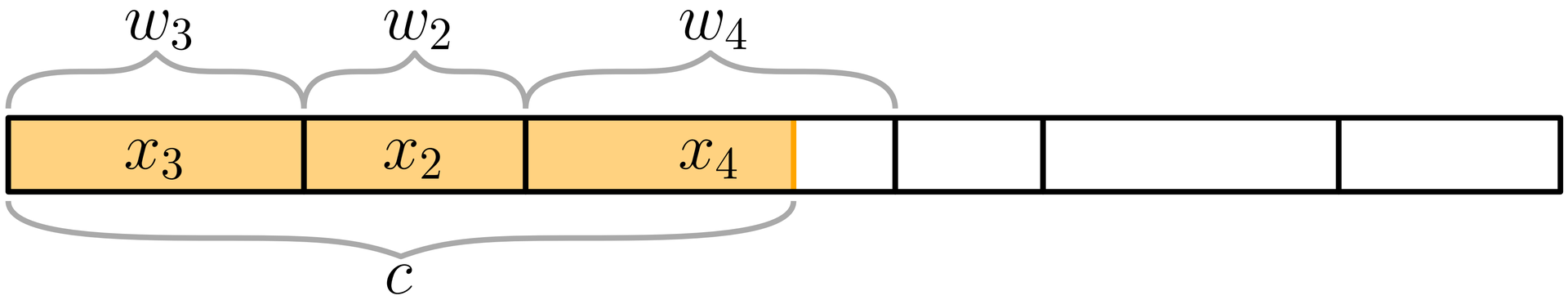
Fractional Greedy Knapsack: Upper Bound

Fractional Greedy Knapsack: Upper Bound

How can we get an *upper* bound? Fractional greedy structure:

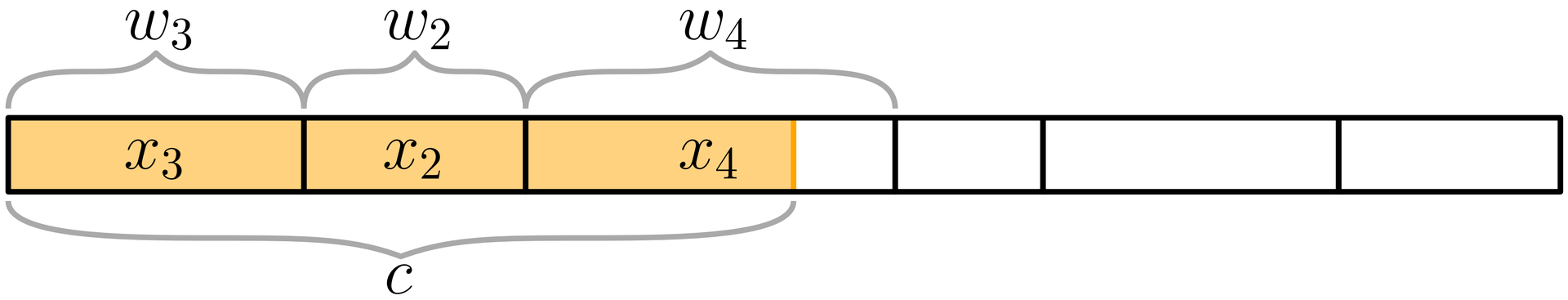
Fractional Greedy Knapsack: Upper Bound

How can we get an *upper* bound? Fractional greedy structure:



Fractional Greedy Knapsack: Upper Bound

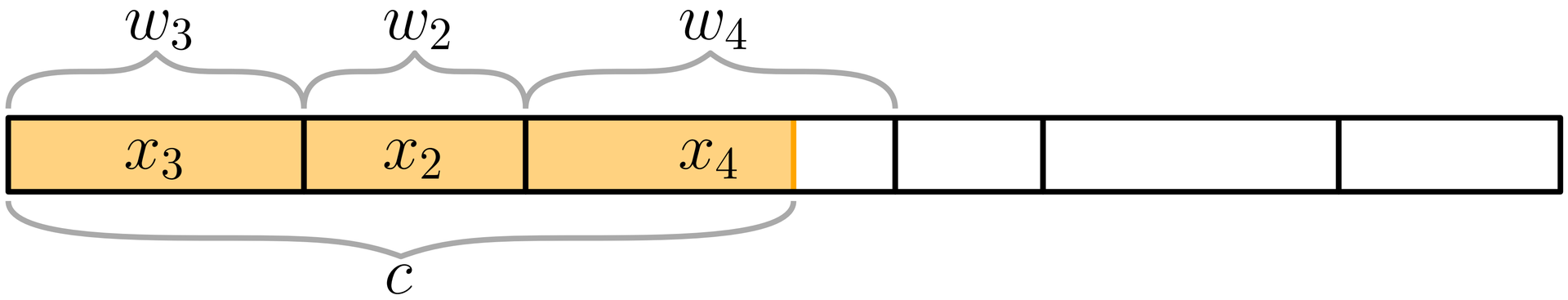
How can we get an *upper* bound? Fractional greedy structure:



Select: $x_3 = 1, x_2 = 1, x_4 = \dfrac{c - w_3 - w_2}{w_4}$, all other $x_i = 0$.

Fractional Greedy Knapsack: Upper Bound

How can we get an *upper* bound? Fractional greedy structure:



Select: $x_3 = 1, x_2 = 1, x_4 = \dfrac{c - w_3 - w_2}{w_4}$, all other $x_i = 0$.

Total price: $P = \sum_{1 \le i \le n} p_i x_i$ is an upper bound on the fractional Knapsack value.

Upper Bound Proof

Upper Bound Proof

Assume, w.l.o.g., $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, else relabel.

Upper Bound Proof

Assume, w.l.o.g., $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, else relabel.

Consider a fractional solution that does not have the greedy structure: there are $i < j$ with $x_i < 1$ and $x_j > 0$.

Upper Bound Proof

Assume, w.l.o.g., $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, else relabel.

Consider a fractional solution that does not have the greedy structure: there are $i < j$ with $x_i < 1$ and $x_j > 0$.

Let $\delta = \min \left\{ w_i(1 - x_i), w_j x_j \right\}$.

Upper Bound Proof

Assume, w.l.o.g., $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, else relabel.

Consider a fractional solution that does not have the greedy structure:
there are $i < j$ with $x_i < 1$ and $x_j > 0$.

Let $\delta = \min \left\{ w_i(1 - x_i), w_j x_j \right\}$.

Idea: reduce the taken weight of x_j by δ and increase the taken weight of x_i by δ,
i.e., $x_i \leftarrow x_i + \delta / w_i$, $x_j \leftarrow x_j - \delta / w_j$; afterwards, either $x_i = 1$ or $x_j = 0$; stays feasible!

Upper Bound Proof

Assume, w.l.o.g., $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$, else relabel.

Consider a fractional solution that does not have the greedy structure:
there are $i < j$ with $x_i < 1$ and $x_j > 0$.

Let $\delta = \min \left\{ w_i(1 - x_i), w_j x_j \right\}$.

Idea: reduce the taken weight of x_j by δ and increase the taken weight of x_i by δ,
i.e., $x_i \leftarrow x_i + \delta / w_i$, $x_j \leftarrow x_j - \delta / w_j$; afterwards, either $x_i = 1$ or $x_j = 0$; stays feasible!

Price change: $\Delta P = \delta / w_i \cdot p_i - \delta / w_j \cdot p_j = \delta \cdot \left(\frac{p_i}{w_i} - \frac{p_j}{w_j} \right) \geq 0.$

Upper Bound Proof

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

➡ Greedy algorithm produces an optimal solution for *fractional Knapsack*.

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

  ➡ Greedy algorithm produces an optimal solution for *fractional Knapsack*.
- Since the solutions to Knapsack are a subset of solutions to fractional Knapsack:

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

  ➡ Greedy algorithm produces an optimal solution for *fractional Knapsack*.
- Since the solutions to Knapsack are a subset of solutions to fractional Knapsack:

  ➡ Greedy's value is an *upper bound* on the value of the optimal Knapsack solution.

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

  ➡ Greedy algorithm produces an optimal solution for *fractional Knapsack*.
- Since the solutions to Knapsack are a subset of solutions to fractional Knapsack:

  ➡ Greedy's value is an *upper bound* on the value of the optimal Knapsack solution.
- We say: Fractional Knapsack *is a* ***relaxation*** of Knapsack.

Upper Bound Proof

- Repeated applications of this create a solution with greedy structure.
- So among all solutions of the same total weight, greedy structure is optimal.
- Maximizing the taken weight (either take all positive items or total weight c):

  ➡ Greedy algorithm produces an optimal solution for *fractional Knapsack*.
- Since the solutions to Knapsack are a subset of solutions to fractional Knapsack:

  ➡ Greedy's value is an *upper bound* on the value of the optimal Knapsack solution.
- We say: Fractional Knapsack *is a* ***relaxation*** of Knapsack.
- This still holds if we fix some x_i to 0 or 1 beforehand.

Approaches

Different approaches:

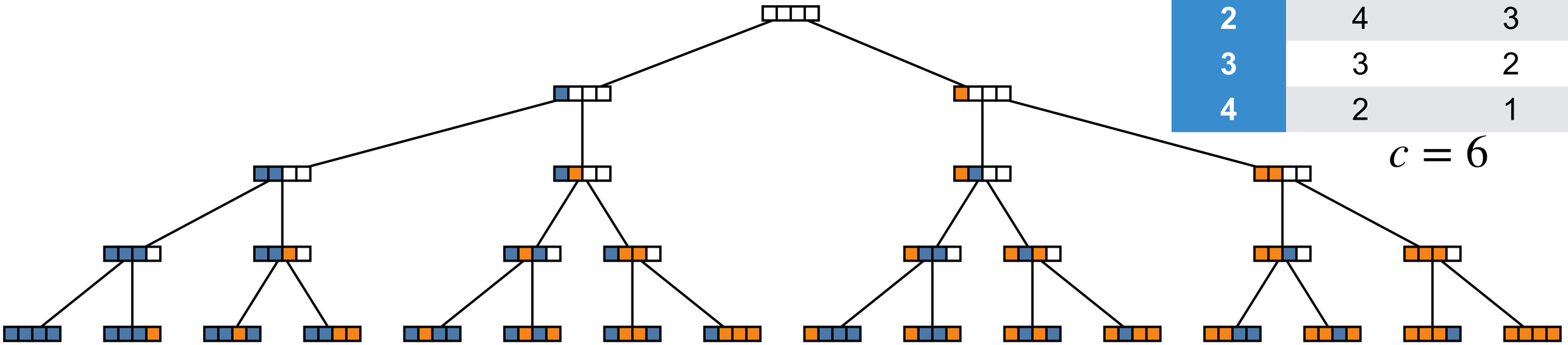
- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Brute Force: Full Search Tree

Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

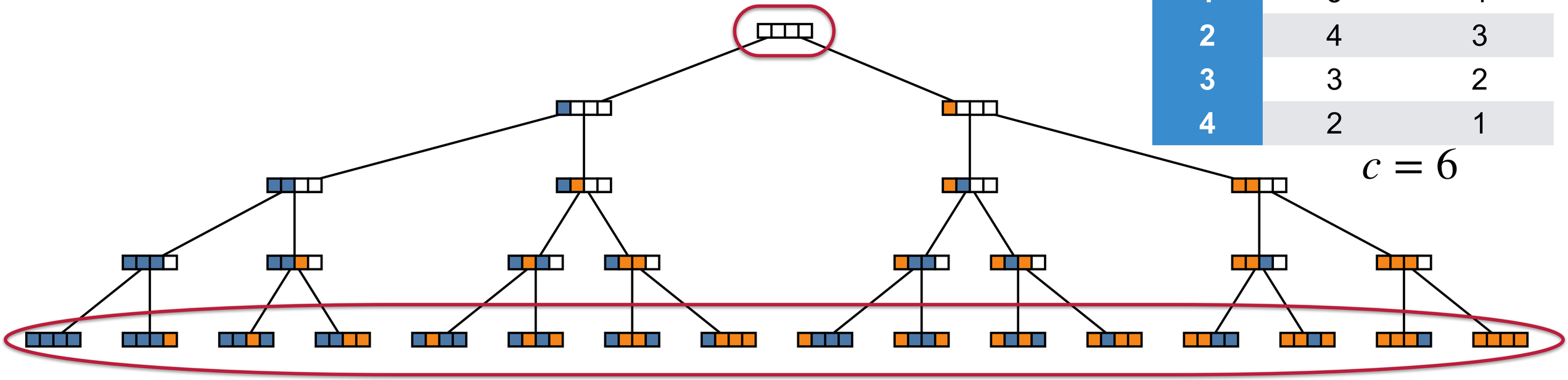
$c = 6$



Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

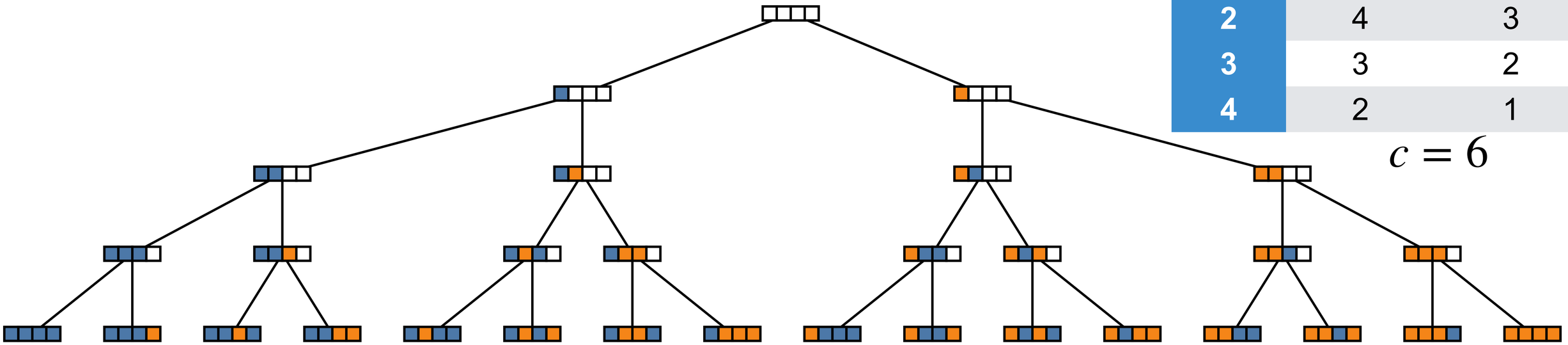
$c = 6$



Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

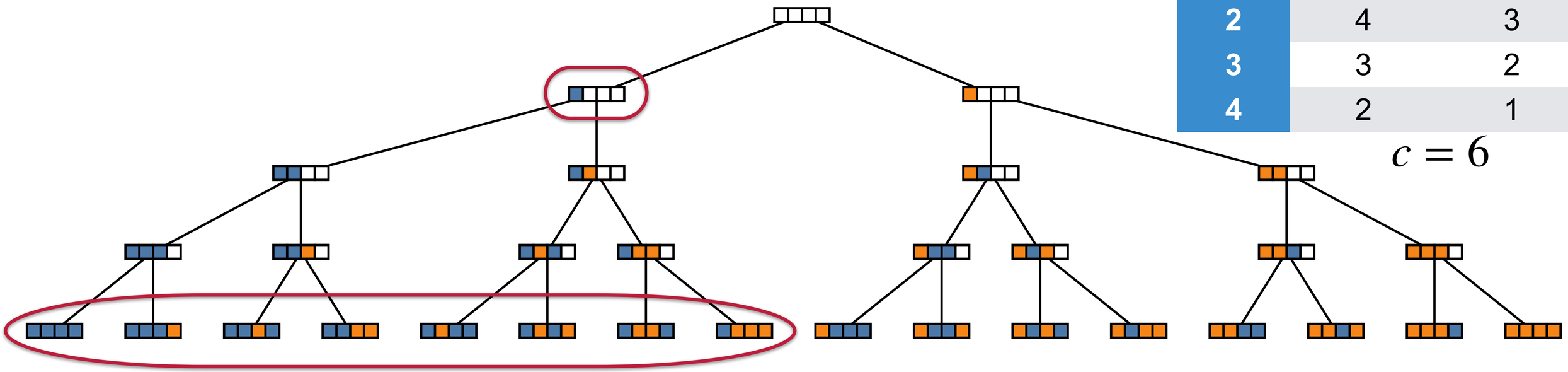
$c = 6$



Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

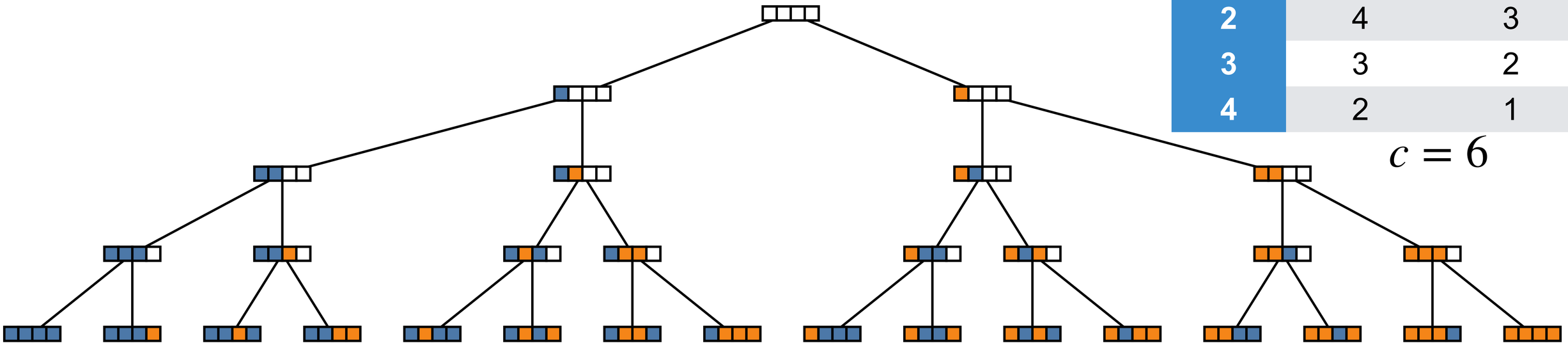
$c = 6$



Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

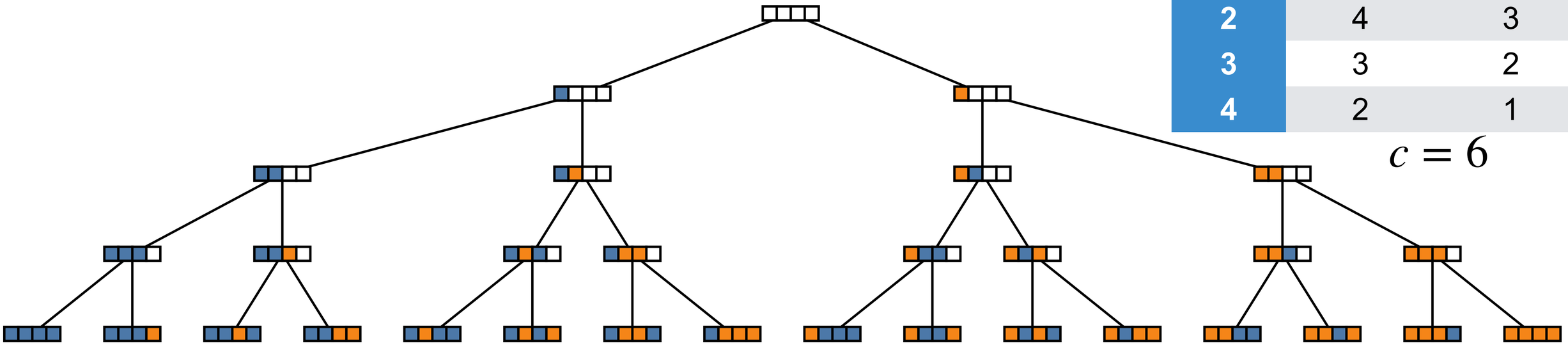
$c = 6$



Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

$c = 6$

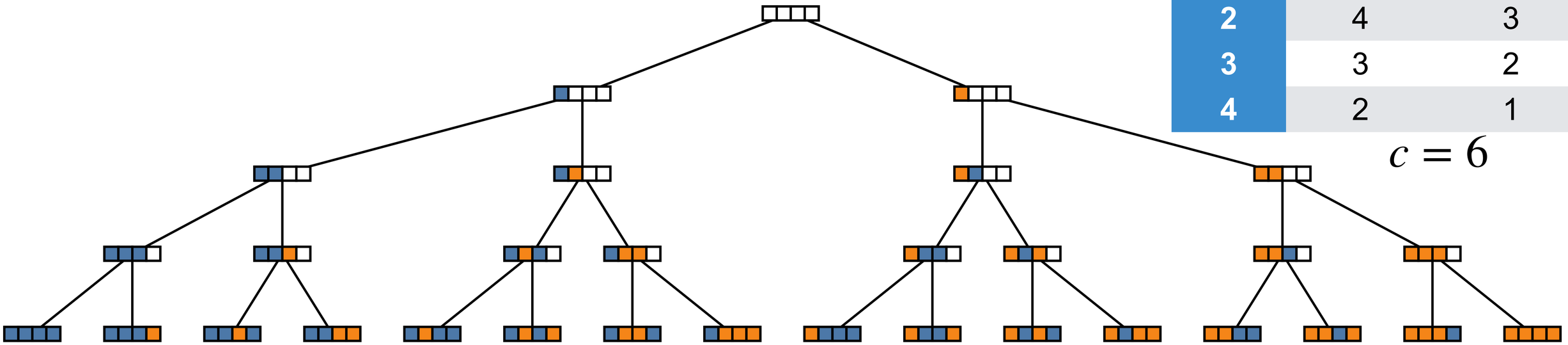


- Full search tree for 4 items: 31 nodes

Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

$c = 6$

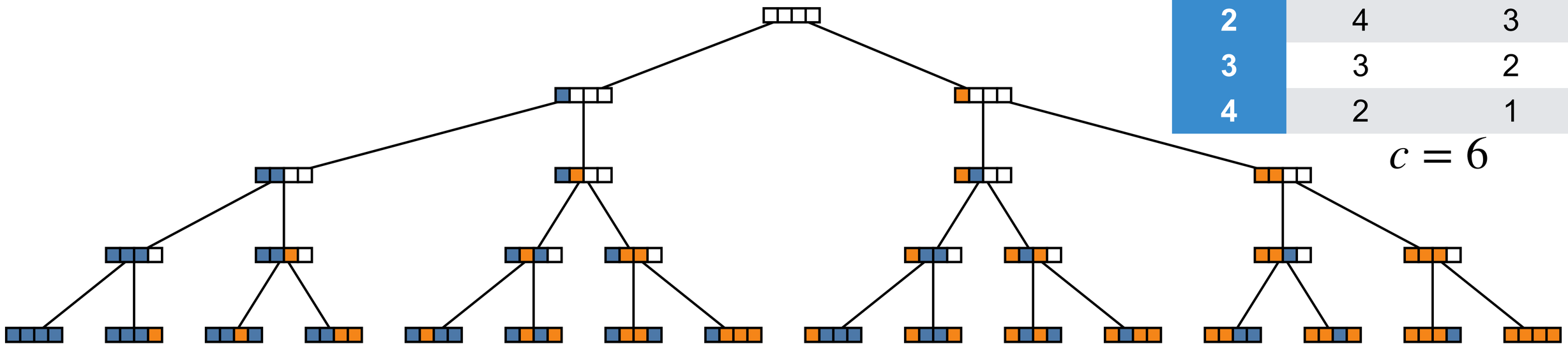


- Full search tree for 4 items: 31 nodes
- Terminology: *partial assignments* (nodes); making *decisions* (edges)

Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

$c = 6$

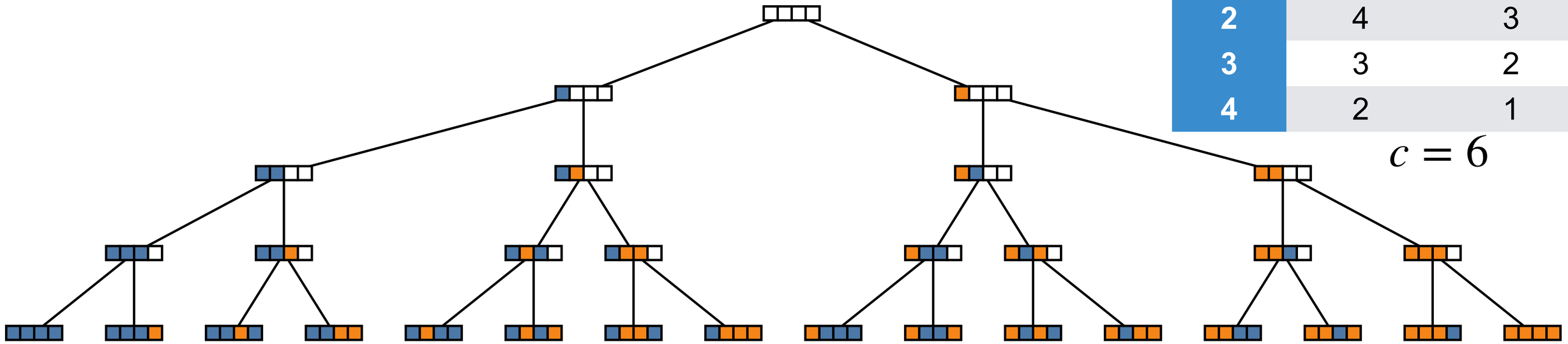


- Full search tree for 4 items: 31 nodes
- Terminology: *partial assignments* (nodes); making *decisions* (edges)
- **Many issues:**

Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

$c = 6$

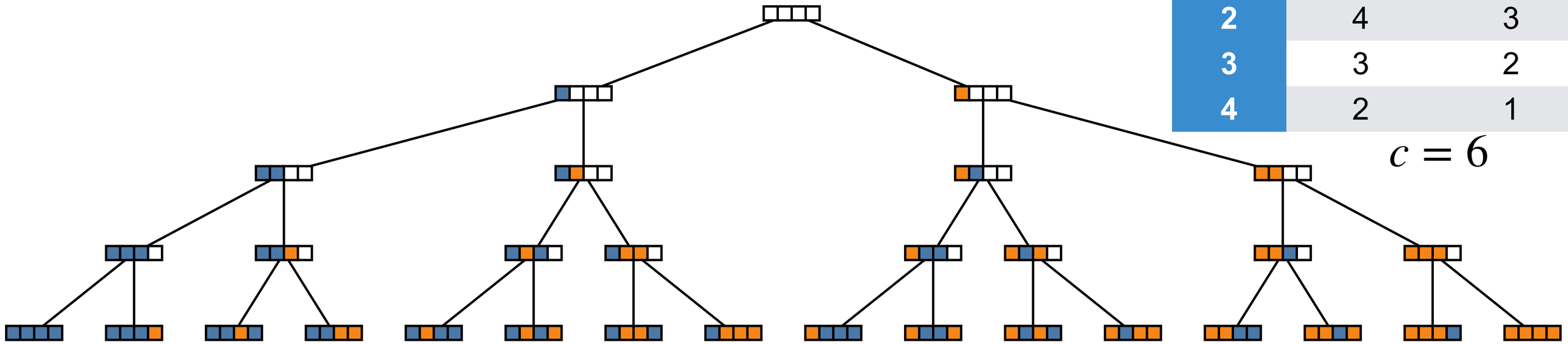


- Full search tree for 4 items: 31 nodes
- Terminology: *partial assignments* (nodes); making *decisions* (edges)
- **Many issues:**
  - Lots of infeasible solutions

Brute Force: Full Search Tree

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

$c = 6$



- Full search tree for 4 items: 31 nodes
- Terminology: *partial assignments* (nodes); making *decisions* (edges)
- **Many issues:**
  - Lots of infeasible solutions
  - Lots of low-quality solutions

Backtracking

$$c = 6$$

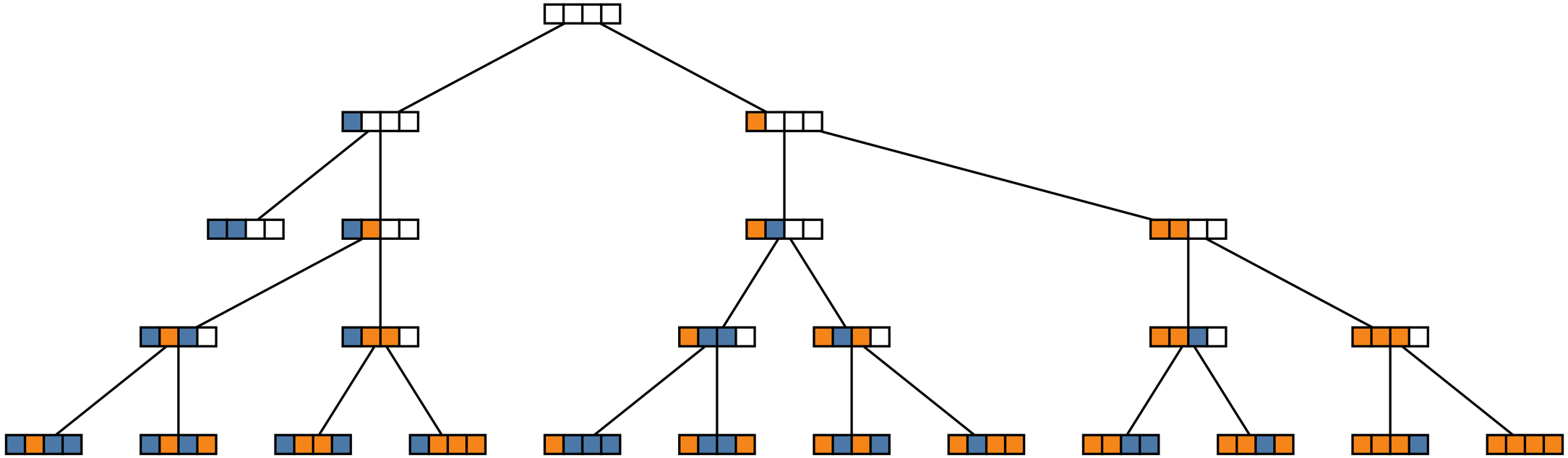
Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Eliminate some infeasible nodes:

$$c = 6$$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Backtracking



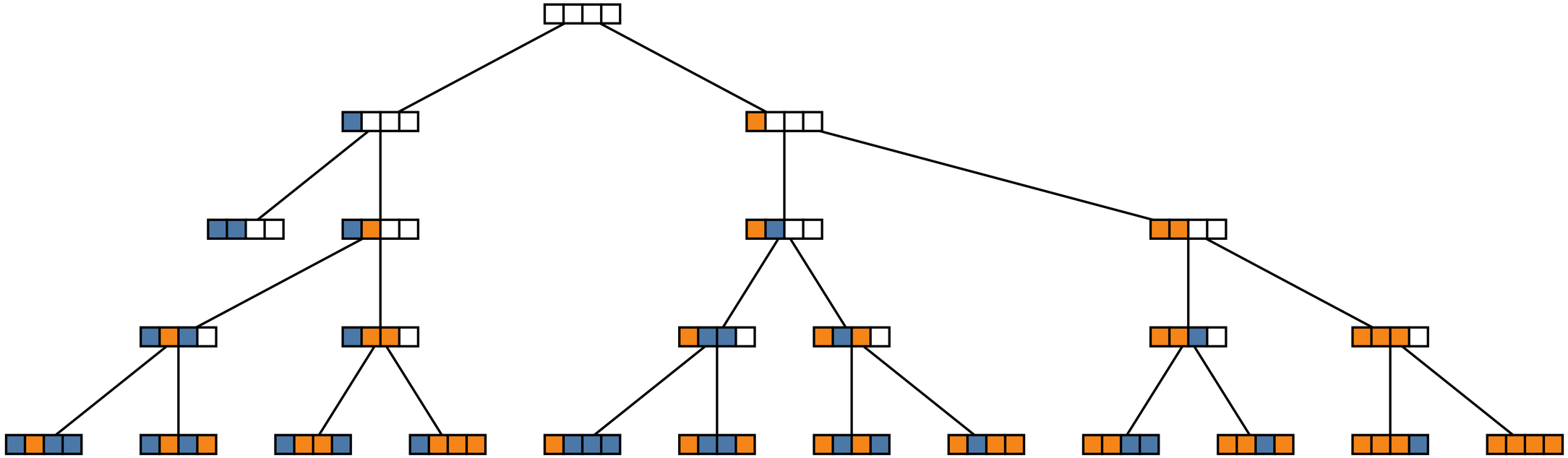
Eliminate some infeasible nodes:

- If the partial solution is already infeasible (too heavy): do not branch any further; discard the node

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Backtracking



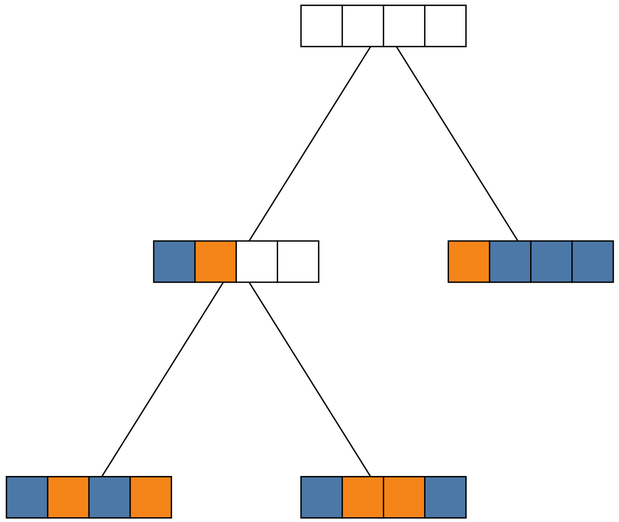
Eliminate some infeasible nodes:

- If the partial solution is already infeasible (too heavy): do not branch any further; discard the node
- Backtracking search tree for 4 items: 31 → 25 nodes

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Backtracking & Constraint Propagation

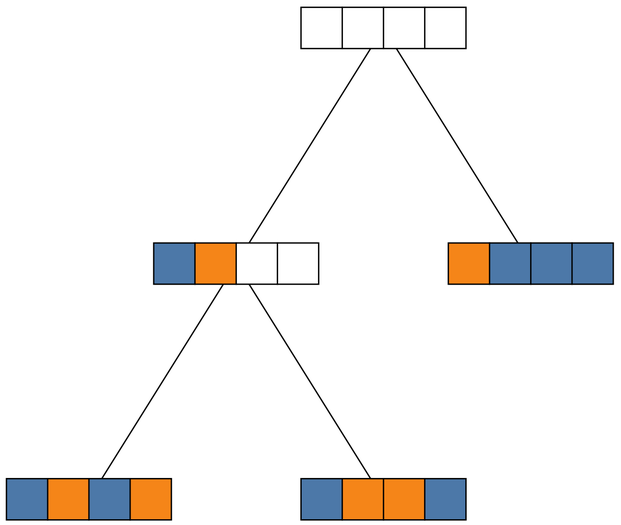


```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Backtracking & Constraint Propagation



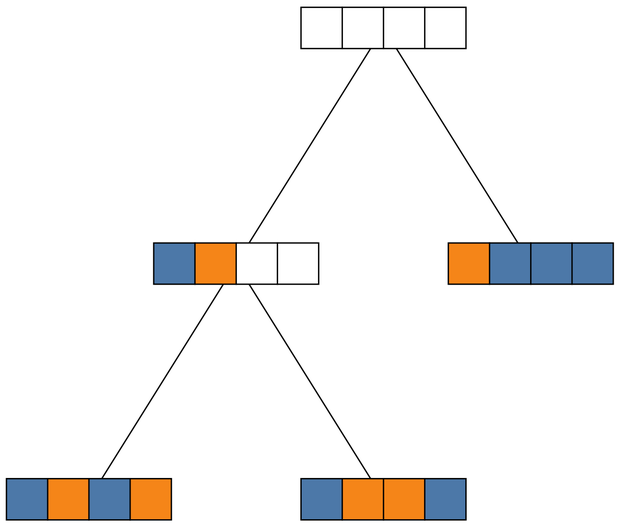
```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Propagation (`propagate`): Compute logical consequences of partial assignments.

Backtracking & Constraint Propagation



```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

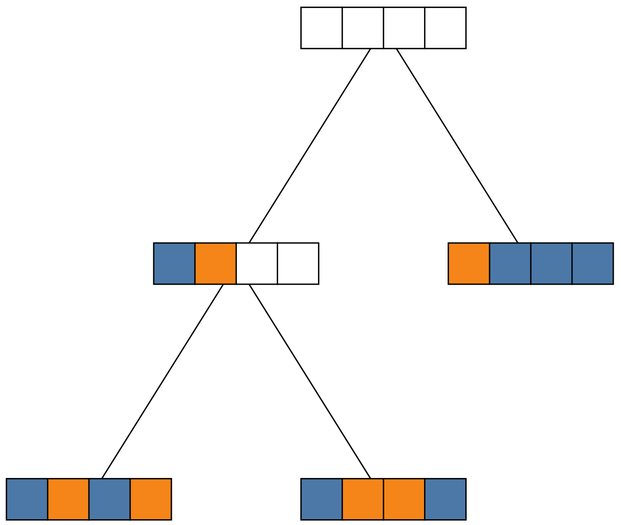
$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Propagation (`propagate`): Compute logical consequences of partial assignments.

Pure constraint propagation: Pure logical consequences of constraints & assignment.

Backtracking & Constraint Propagation



```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

$c = 6$

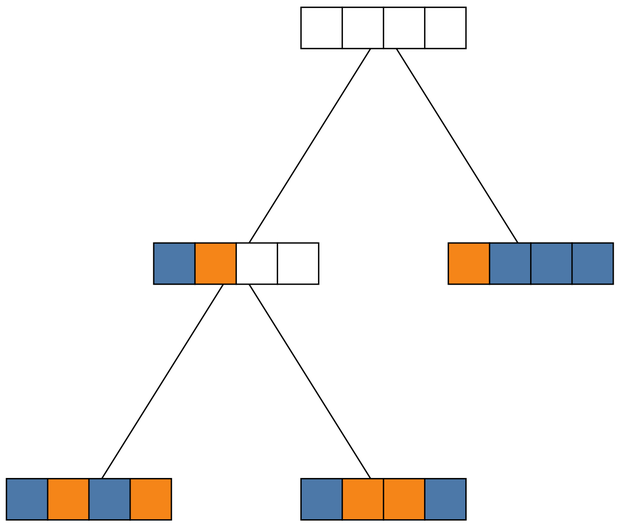
Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Propagation (`propagate`): Compute logical consequences of partial assignments.

Pure constraint propagation: Pure logical consequences of constraints & assignment.

Knapsack: fix to false all items that do not fit into the container, given the items set to true.

Backtracking & Constraint Propagation



```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

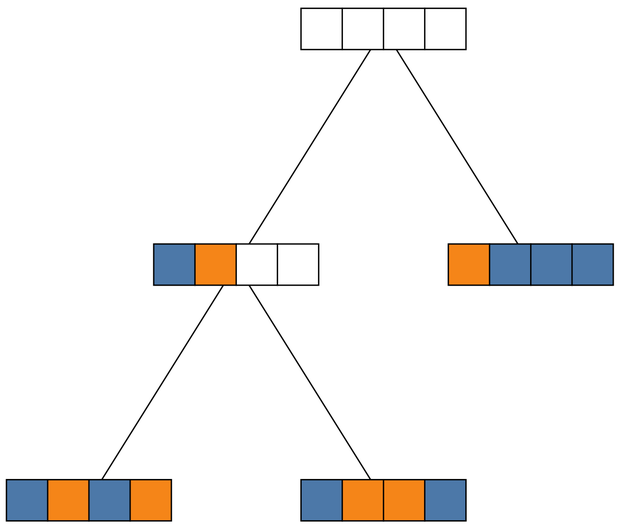
Propagation (`propagate`): Compute logical consequences of partial assignments.

Pure constraint propagation: Pure logical consequences of constraints & assignment.

Knapsack: fix to false all items that do not fit into the container, given the items set to true.

Objective-based propagation: Additionally, use the objective.

Backtracking & Constraint Propagation



```
backtrack(partial):
  if infeasible(partial):
    return
  partial = propagate(partial)
  if index = unfixed(partial):
    partial[index] = True
    backtrack(partial)
    partial[index] = False
    backtrack(partial)
  else:
    record(partial)
```

$c = 6$

Item	Price p	Weight w
1	5	4
2	4	3
3	3	2
4	2	1

Propagation (`propagate`): Compute logical consequences of partial assignments.

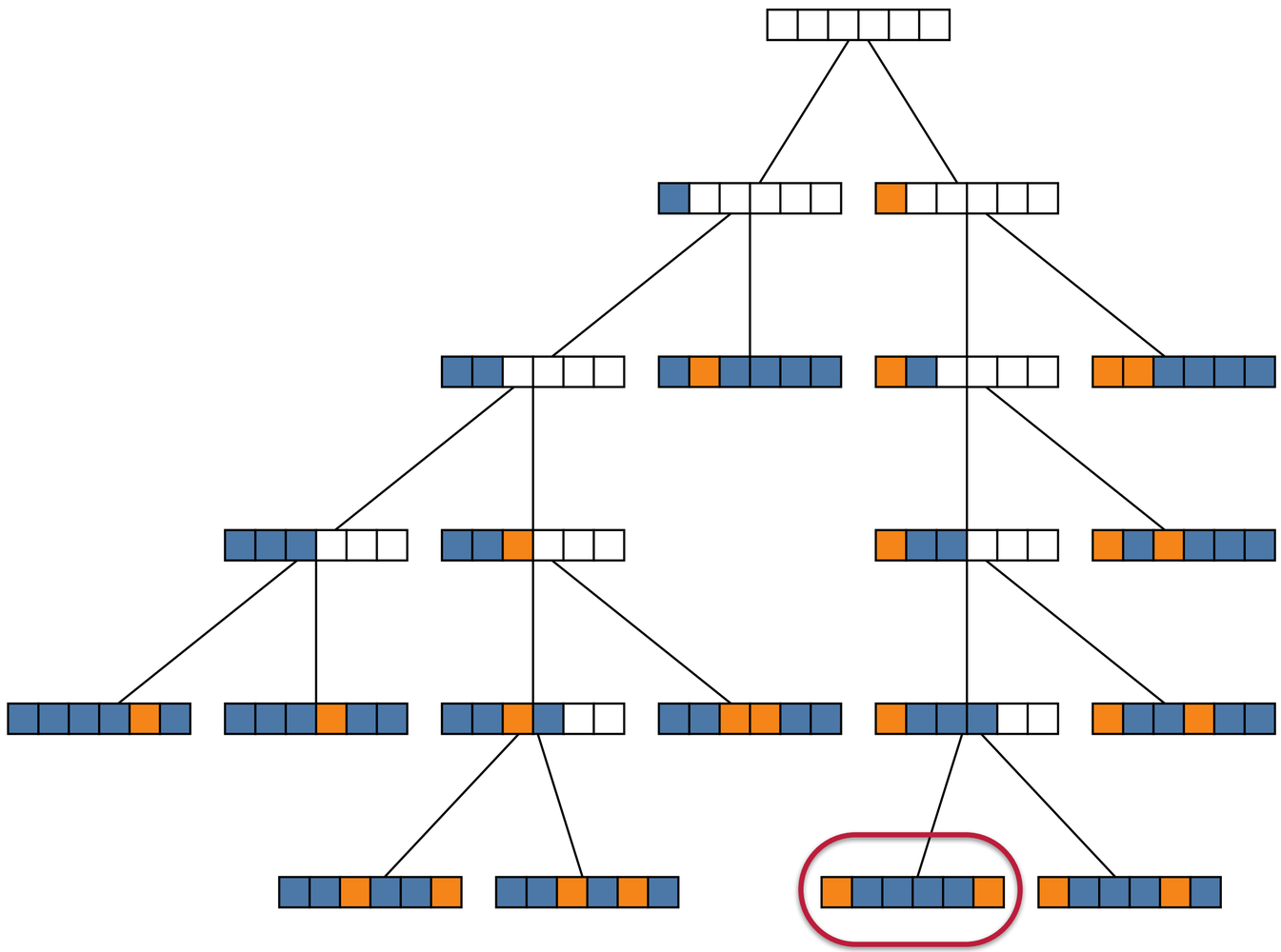
Pure constraint propagation: Pure logical consequences of constraints & assignment.

Knapsack: fix to false all items that do not fit into the container, given the items set to true.

Objective-based propagation: Additionally, use the objective.

Knapsack: fix to true all unfixed items with positive price if they all fit into the container.

Harder Instance — Backtracking & Constraint Propagation



OPT: 77 $\{2,3,4,5\}$

$$c = 19$$

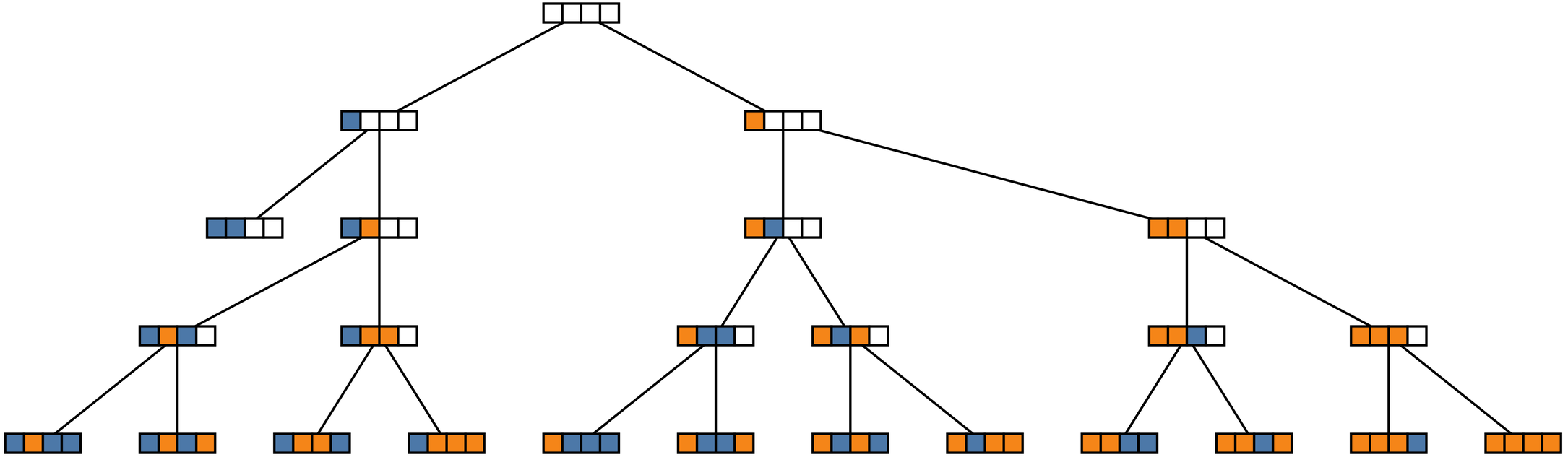
Item	Price p	Weight w
1	10	1
2	20	7
3	16	3
4	20	5
5	21	4
6	8	3

Remaining issues:
Many suboptimal nodes
Repeated subtrees

Backtracking & Constraint Propagation: Conflict Analysis

In propagation/infeasibility checking:

- Analyze *why* infeasibilities occur
- Learn more general rules than ‘that partial assignment is infeasible’
- More on that later: CDCL SAT Solvers, CP-SAT



Approaches

Different approaches:

- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Branch & Bound: Key Insight

Branch & Bound: Key Insight

Backtracking:

- Throw away subtrees that cannot contain *any* valid solution

Branch & Bound: Key Insight

Backtracking:

- Throw away subtrees that cannot contain *any* valid solution
- Objective-based propagation: already does a little more

Branch & Bound: Key Insight

Backtracking:

- Throw away subtrees that cannot contain *any* valid solution
- Objective-based propagation: already does a little more
- **Question:** When can we safely discard a whole subtree?

Branch & Bound: Key Insight

Backtracking:

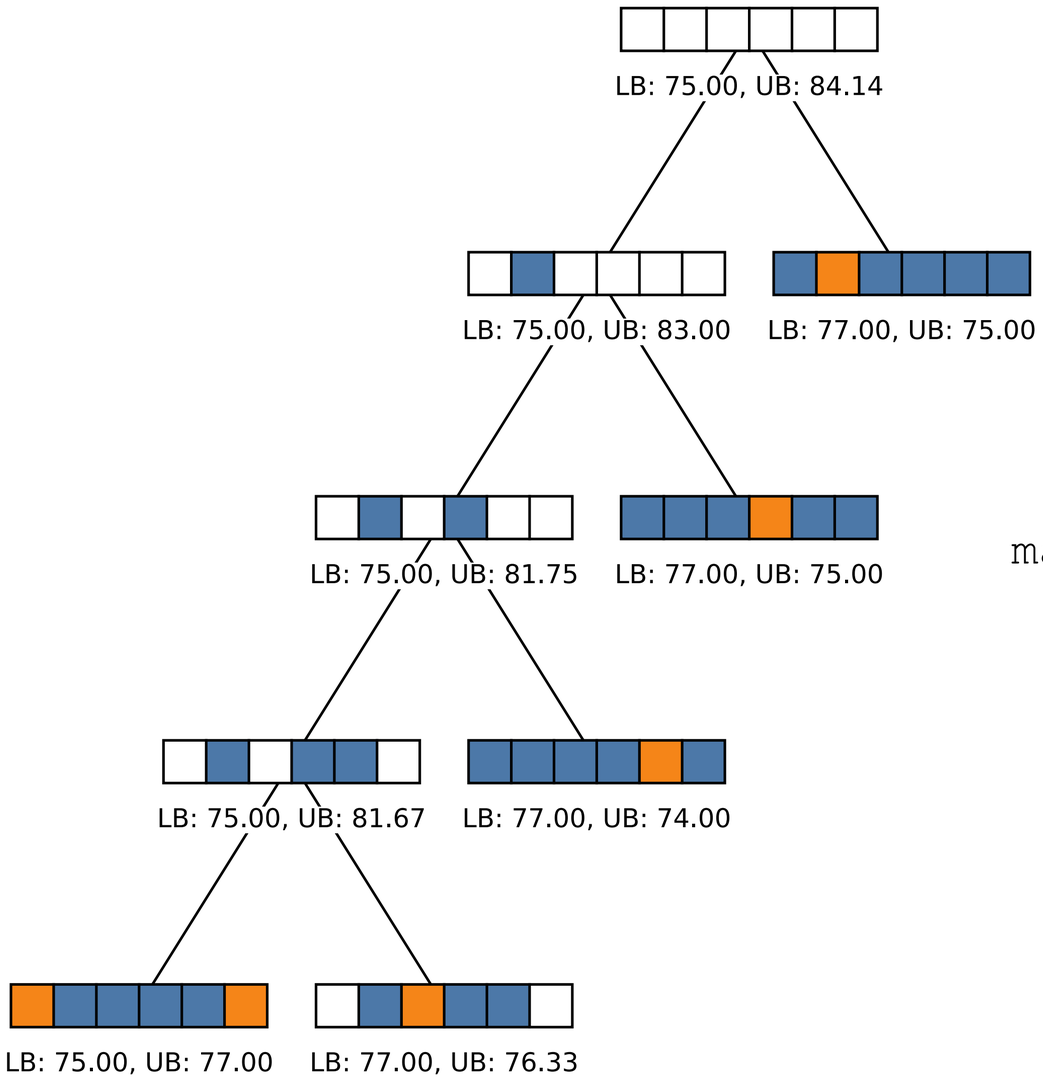
- Throw away subtrees that cannot contain *any* valid solution
- Objective-based propagation: already does a little more
- **Question:** When can we safely discard a whole subtree?
- When it cannot contain any *better* valid solution than one we already have

Branch & Bound: Key Insight

Backtracking:

- Throw away subtrees that cannot contain *any* valid solution
- Objective-based propagation: already does a little more
- **Question:** When can we safely discard a whole subtree?
- When it cannot contain any *better* valid solution than one we already have
- Try and track (at sustainable effort) whether each subtree can still have a better solution!

Branch & Bound: Pruning



$c = 19$

Item	Price p	Weight w
1	10	1
2	20	7
3	16	3
4	20	5
5	21	4
6	8	3

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus
- Breadth first: exploration

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus
- Breadth first: exploration

- Problem-specific

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus
- Breadth first: exploration

- Problem-specific
- UB always needed

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus
- Breadth first: exploration

- Problem-specific
- UB always needed
- LB = UB for full solutions

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Exploration order:

- Depth first: memory efficient
- Best first: UB focus
- Breadth first: exploration

- Problem-specific
- UB always needed
- LB = UB for full solutions

Branch (variable) selection: Only one reasonable choice in KP

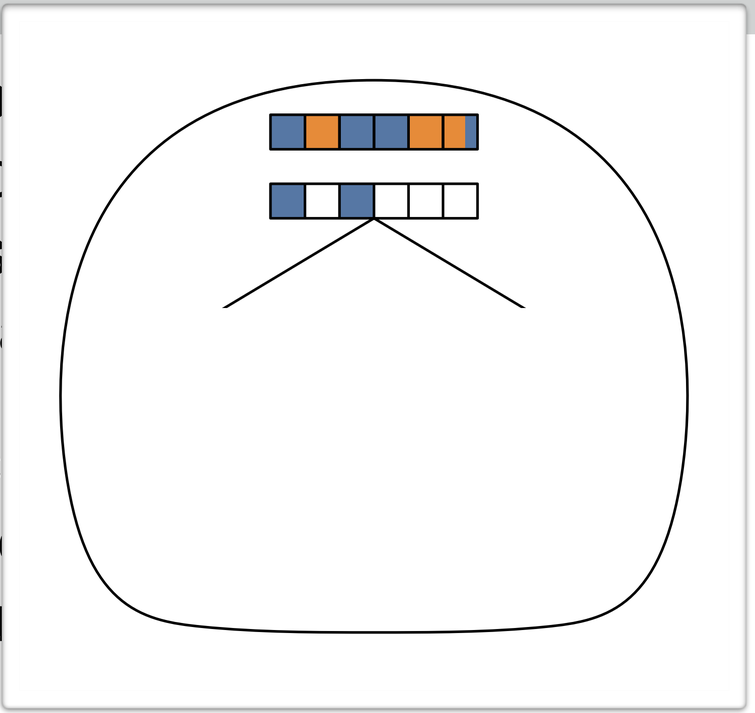
Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre

- Problem-spec
- UB always ne
- LB = UB for fu



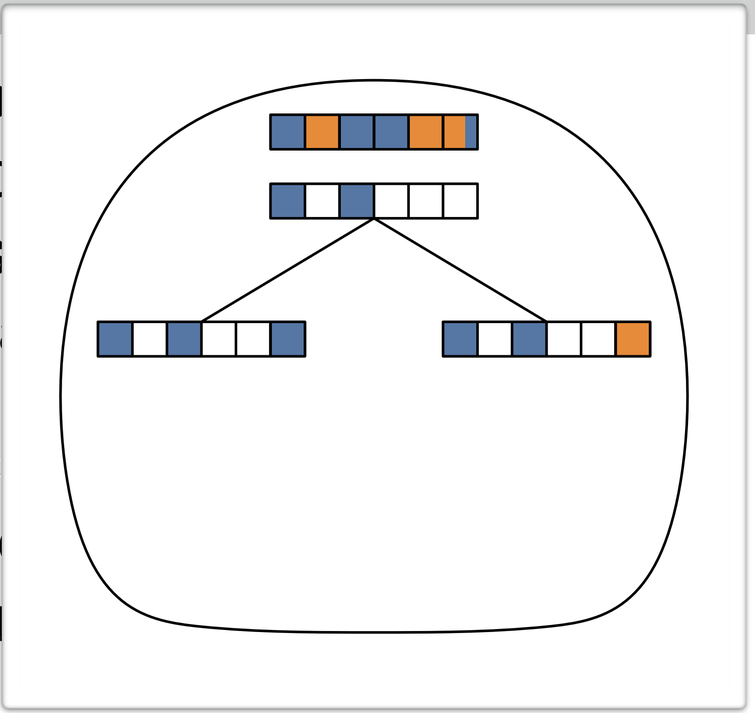
Branch (variable) selection: Only one reasonable choice in KP

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

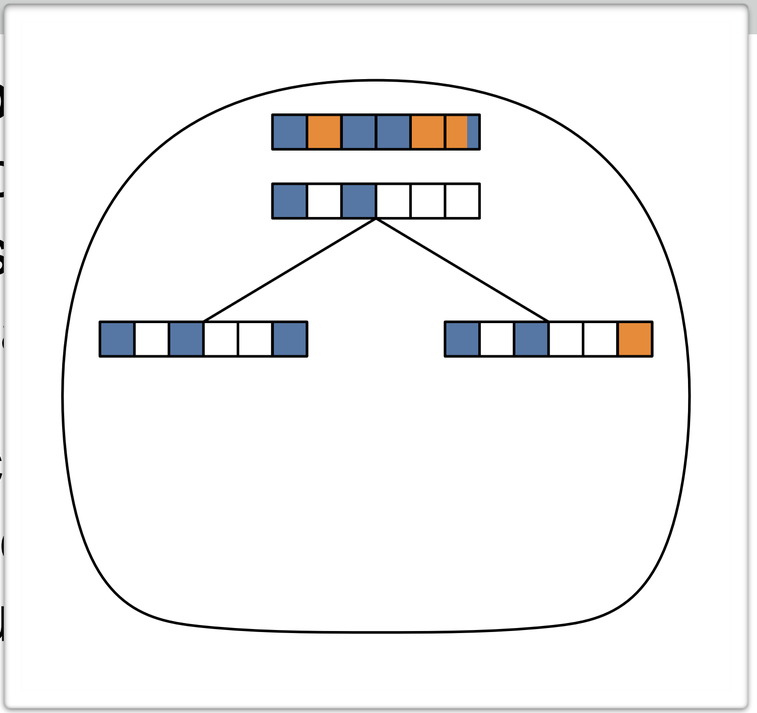
Branch (variable) selection: Only one reasonable choice in KP

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

Branch (variable) selection: Only one reasonable choice in KP

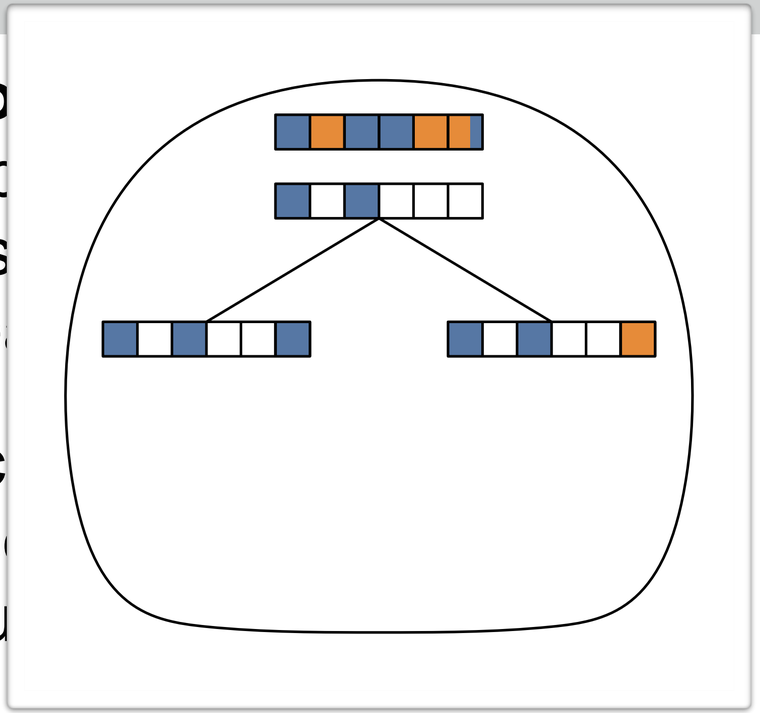
- Which reduces the tree most? → we cannot know; heuristic choice

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

Branch (variable) selection: Only one reasonable choice in KP

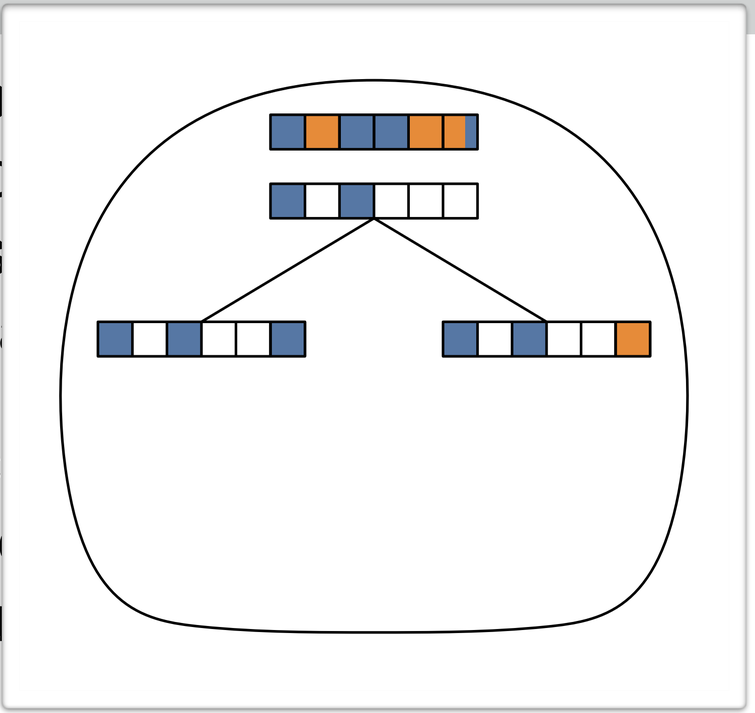
- Which reduces the tree most? → we cannot know; heuristic choice
- Strong branching: try multiple variables, compute bounds on both sides

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

Branch (variable) selection: Only one reasonable choice in KP

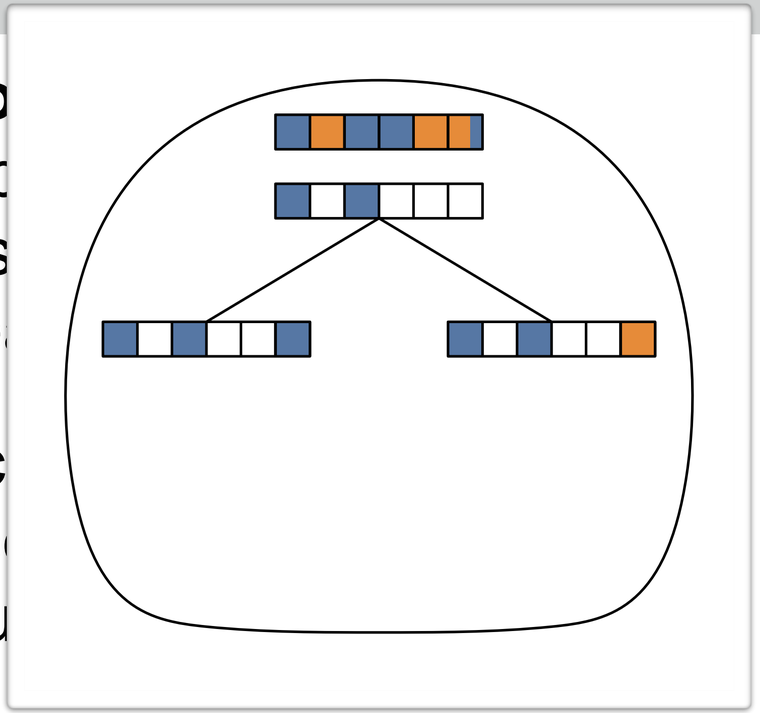
- Which reduces the tree most? → we cannot know; heuristic choice
- Strong branching: try multiple variables, compute bounds on both sides
- Pseudocost branching (average bound improvement per unit change)

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

Branch (variable) selection: Only one reasonable choice in KP

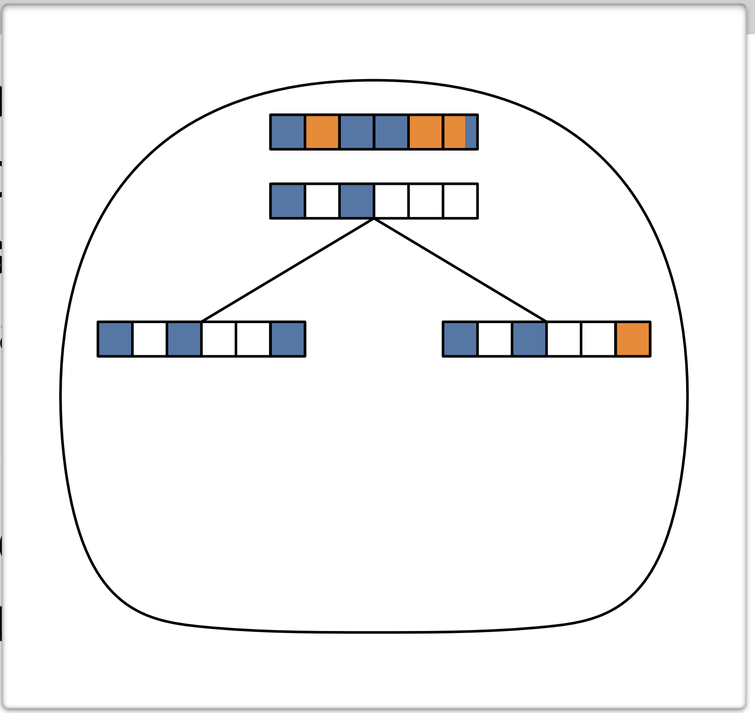
- Which reduces the tree most? → we cannot know; heuristic choice
- Strong branching: try multiple variables, compute bounds on both sides
- Pseudocost branching (average bound improvement per unit change)
- Reliability branching

Branch & Bound: Open Factors

```
max_bnb(problem):
  lb = heuristic_solution(problem)
  open_nodes = [empty_partial(problem)]
  while node = select_and_pop(open_nodes):
    if not partial = propagate_and_check(node):
      continue  # infeasible
    node_lb, node_ub = bound(partial)
    if node_lb and node_lb > lb:
      lb = node_lb
    if node_ub <= lb:
      continue  # prune
    open_nodes += branch(partial)
```

Explo

- Dep
- Bes
- Bre



- Problem-spec
- UB always ne
- LB = UB for fu

Branch (variable) selection: Only one reasonable choice in KP

- Which reduces the tree most? → we cannot know; heuristic choice
- Strong branching: try multiple variables, compute bounds on both sides
- Pseudocost branching (average bound improvement per unit change)
- Reliability branching
- Simple scores (distance to integrality), Hybrid, ML, conflict-based, …

Approaches

Different approaches:

- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Extension: Branch & Cut

Linear program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$\in P$$

Extension: Branch & Cut

Linear program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$\in P$

Binary program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \{0,1\}$$

NP-complete

Extension: Branch & Cut

Linear program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$\in P$

Binary program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \{0,1\}$$

NP-complete

Integer program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \mathbb{Z}$$

NP-complete

Extension: Branch & Cut

Linear program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$\in P$

Binary program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \{0,1\}$$

NP-complete

Integer program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \mathbb{Z}$$

NP-complete

Mixed Integer Program (MIP): mixing integer/real variables

Extension: Branch & Cut

Linear program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$\in P$

Binary program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \{0,1\}$$

NP-complete

Integer program

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \in \mathbb{Z}$$

NP-complete

Mixed Integer Program (MIP): mixing integer/real variables

Most common implementation of `bound`**:**

Solve the linear relaxation (drop $x_i \in \mathbb{Z}$, $x_i \in \{0,1\} \rightarrow x_i \in [0,1]$; turns MIP into LP)

Extension: Branch & Cut

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Extension: Branch & Cut

Knapsack (Bin. Program)

```python
def solve_knapsack_gurobi(instance: KnapsackProblem):
    model = grb.Model()
    n = instance.item_count
    p = instance.prices
    w = instance.weights
    c = instance.capacity
    x = [model.addVar(vtype=grb.GRB.BINARY, obj=p[i])
         for i in range(n)]
    model.ModelSense = grb.GRB.MAXIMIZE
    model.addConstr(grb.quicksum(x[i] * w[i] for i in range(n)) <= c)
    model.optimize()
    values = [x_i.X for x_i in x]
    return KnapsackSolution(
        items_included=[i for i in range(n) if values[i] > 0.5],
        total_price=model.ObjVal
    )
```

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

BP formulation of Knapsack: equivalent to Knapsack (*Formulation* of Knapsack as BP)

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

BP formulation of Knapsack: equivalent to Knapsack (*Formulation* of Knapsack as BP)

LP relaxation: equivalent to fractional greedy Knapsack

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

BP formulation of Knapsack: equivalent to Knapsack (*Formulation* of Knapsack as BP)

LP relaxation: equivalent to fractional greedy Knapsack

Improving `bound`**:** find additional *valid* constraints

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

BP formulation of Knapsack: equivalent to Knapsack (*Formulation* of Knapsack as BP)

LP relaxation: equivalent to fractional greedy Knapsack

Improving `bound`**:** find additional *valid* constraints
satisfied by all integer solutions

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

Given: optimal, non-integral solution X to the linear relaxation

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

Given: optimal, non-integral solution X to the linear relaxation

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

Given: optimal, non-integral solution X to the linear relaxation

Goal: linear constraint $\sum_i a_i x_i \leq b$ violated by X, but satisfied by all integral solutions

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

$$x_3 + x_2 + x_4 \leq ?$$

Given: optimal, non-integral solution X to the linear relaxation

Goal: linear constraint $\sum_i a_i x_i \leq b$ violated by X, but satisfied by all integral solutions

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

$$x_3 + x_2 + x_4 \leq 2$$

Given: optimal, non-integral solution X to the linear relaxation

Goal: linear constraint $\sum_i a_i x_i \leq b$ violated by X, but satisfied by all integral solutions

Extension: Branch & Cut

Knapsack (Bin. Program)

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$x_i \in \{0,1\}$$

NP-complete

Linear Relaxation

$$\max \sum_i p_i x_i \text{ s.t.}$$

$$\sum_i w_i x_i \leq c$$

$$0 \leq x_i \leq 1$$

$\in P$

$$x_3 + x_2 + x_4 \leq 2$$

Given: optimal, non-integral solution X to the linear relaxation

Goal: linear constraint $\sum_i a_i x_i \leq b$ violated by X, but satisfied by all integral solutions

Possible cuts: $\sum_{i \in C} x_i \leq |C| - 1$ for some C that does not fit into the Knapsack

Extension: Branch & Cut — Simple Greedy Separation

Extension: Branch & Cut — Simple Greedy Separation

Idea: take relaxed (fractional) solution x^*, order by non-increasing x_i^*

Extension: Branch & Cut — Simple Greedy Separation

Idea: take relaxed (fractional) solution x^*, order by non-increasing x_i^*

In order: construct D in order until $\sum_{i \in D} w_i > c$; if $\sum_{i \in D} x_i^* > |D| - 1$, add $\sum_{i \in D} x_i^* \leq |D| - 1$

Extension: Branch & Cut — Simple Greedy Separation

Idea: take relaxed (fractional) solution x^*, order by non-increasing x_i^*

In order: construct D in order until $\sum_{i \in D} w_i > c$; if $\sum_{i \in D} x_i^* > |D| - 1$, add $\sum_{i \in D} x_i^* \leq |D| - 1$

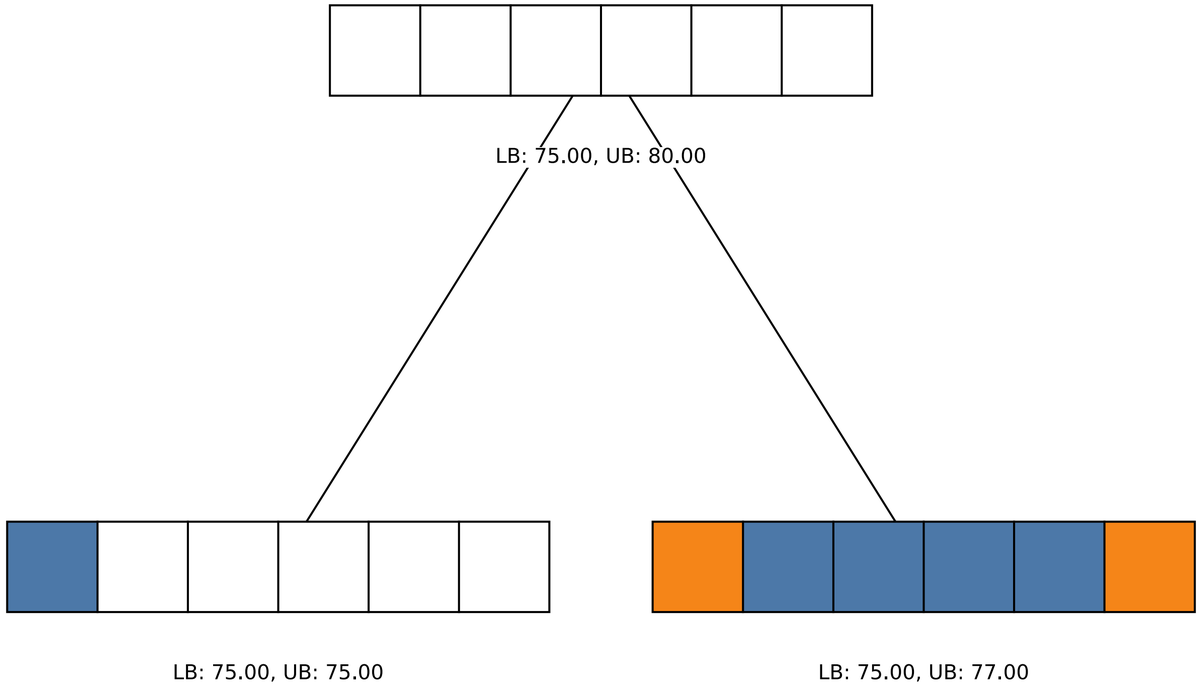
If constraint added: re-solve relaxation; **otherwise:** accept bound, branch if needed

Extension: Branch & Cut — Simple Greedy Separation

Idea: take relaxed (fractional) solution x^*, order by non-increasing x_i^*

In order: construct D in order until $\sum_{i \in D} w_i > c$; if $\sum_{i \in D} x_i^* > |D| - 1$, add $\sum_{i \in D} x_i^* \leq |D| - 1$

If constraint added: re-solve relaxation; **otherwise:** accept bound, branch if needed



Approaches

Different approaches:

- Dynamic programming
- Greedy heuristic & fractional greedy upper bound
- Backtracking & Constraint Propagation
- Branch & Bound
- Branch & Cut
- Metaheuristics

Metaheuristics

Why heuristics?

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

High Level Types of Metaheuristics

Metaheuristics

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

High Level Types of Metaheuristics

- Constructive heuristics

Metaheuristics

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

High Level Types of Metaheuristics

- Constructive heuristics
- Local search and variants

Metaheuristics

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

High Level Types of Metaheuristics

- Constructive heuristics
- Local search and variants
- Population-based algorithms

Metaheuristics

Why heuristics?

- Finding initial solution for pruning → sometimes built into solvers
- Instances too large, problem too nasty → exact search solvers do not scale

High Level Types of Metaheuristics

- Constructive heuristics
- Local search and variants
- Population-based algorithms
- These cover most (meta)heuristics, but there are other approaches

Constructive Heuristics

Constructive Heuristics

- Start with an empty or partial solution

Constructive Heuristics

- Start with an empty or partial solution
- Extend (e.g., greedily)

Constructive Heuristics

- Start with an empty or partial solution
- Extend (e.g., greedily)
- Possibly randomized (e.g., using restricted candidate lists, RCLs)

Constructive Heuristics

- Start with an empty or partial solution
- Extend (e.g., greedily)
- Possibly randomized (e.g., using restricted candidate lists, RCLs)
- Possibly repeated (different random decisions, different parameters, different starts)

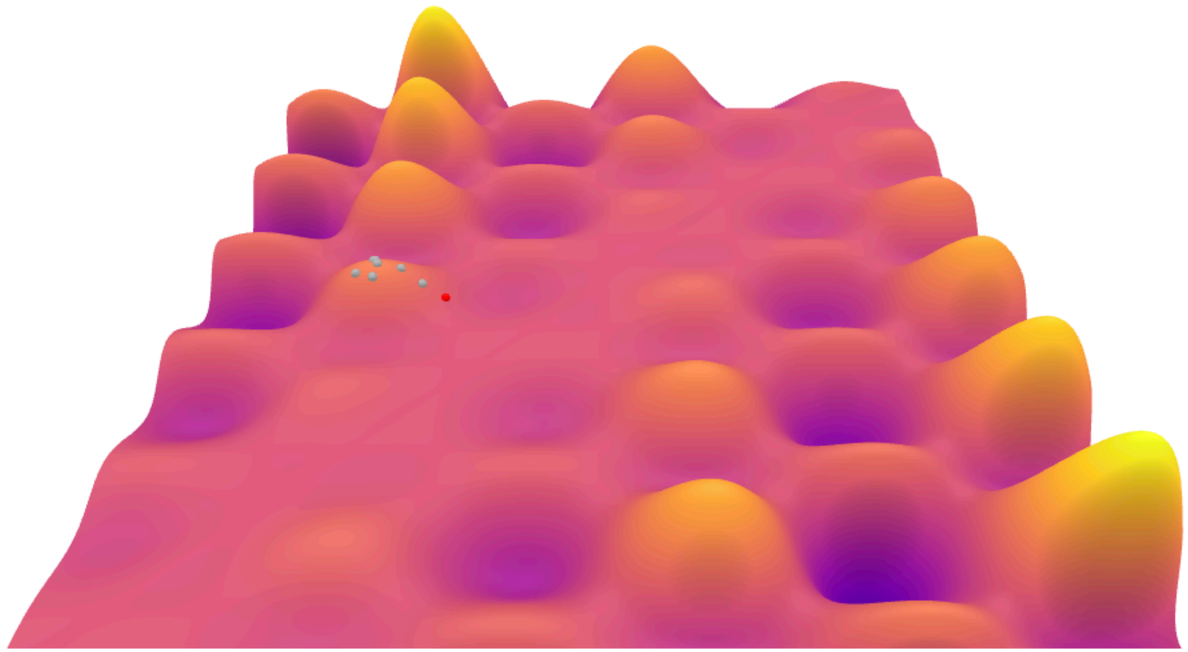
Constructive Heuristics

- Start with an empty or partial solution
- Extend (e.g., greedily)
- Possibly randomized (e.g., using restricted candidate lists, RCLs)
- Possibly repeated (different random decisions, different parameters, different starts)
- Possibly enhanced by local search

Constructive Heuristics

- Start with an empty or partial solution
- Extend (e.g., greedily)
- Possibly randomized (e.g., using restricted candidate lists, RCLs)
- Possibly repeated (different random decisions, different parameters, different starts)
- Possibly enhanced by local search
- Typically fast; trade-off between number of solutions and chance of good solution

Local Search



General idea:

- Starting from a solution S
- Consider neighboring solutions S'
- Stay at S or move to some neighbor S'
- Repeat until some termination criterion

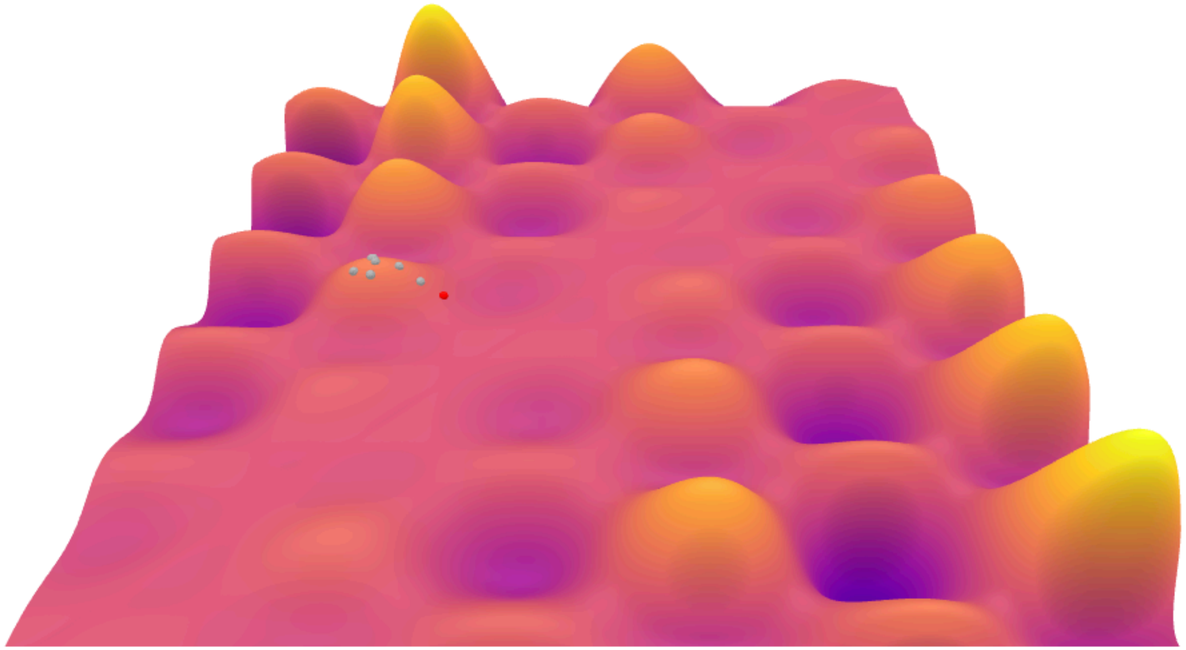
Main issue:

- Local optima

Examples:

- 2-OPT and other simple improvements
- Simulated annealing
- Large Neighborhood Search (LNS)
- Limited Discrepancy Search
- …

Exploration vs. Exploitation



Exploration:

- How good are we at exploring all relevant parts of the search space?

Exploitation:

- How good are we at improving solutions to a local optimum?

Balance: usually start more explorative/broad/diverse, then focus/intensify

Population-based Heuristics

General idea:

- Do not maintain a single solution
- Instead, maintain a population of solutions
- In each iteration (generation), create a new population from the old
- Typically involves combining solutions
- Typically, prefer better solutions
- Typically, some mechanism encourages diversity

Examples:

- Genetic algorithms
- Memetic algorithms
- Particle swarms
- …

Key Ideas

- Tree search idea (partial assignments, connection to solution space)
- Relaxation
- Constraint propagation
- Eliminate nodes/branches that have no solution (backtracking)
- Eliminate nodes/branches that cannot have a better solution (branch & bound)
- Heuristic initial solutions can be worthwhile (better pruning)
- Strengthening bounds can be worthwhile (branch & cut)
- Metaheuristics often are the best practical approach (scalability)
- They can still use exact methods (we will see this later)