Backtracking & Branch-and-Bound

Desain & Analisis Algoritma

Fakultas Ilmu Komputer

Universitas Indonesia

Compiled by **Alfan F. Wicaksono** from multiple sources

Credits

- **Materi DAA Pak Yugo K. Isal**

Review: 0-1 Knapsack Problem

- Greedy solution?
  - Not optimal
- Dynamic Programming solution?
  - **O(nW)** – but not polynomial in input size (**pseudo-polynomial**)
  - The DP solution does not work when an item weights **are not** described with integers.
- Any better solution?
  - No one has ever found an algorithm whose worst-case time complexity is better than exponential.
  - But no one has proved that a polynomial solution **does not** exist.

Review: 0-1 Knapsack Problem

We have a situation where Dynamic Programming does not apply.
That's when our problem cannot be described with integers.

Shall we use **brute-force** solution instead?

W = 10

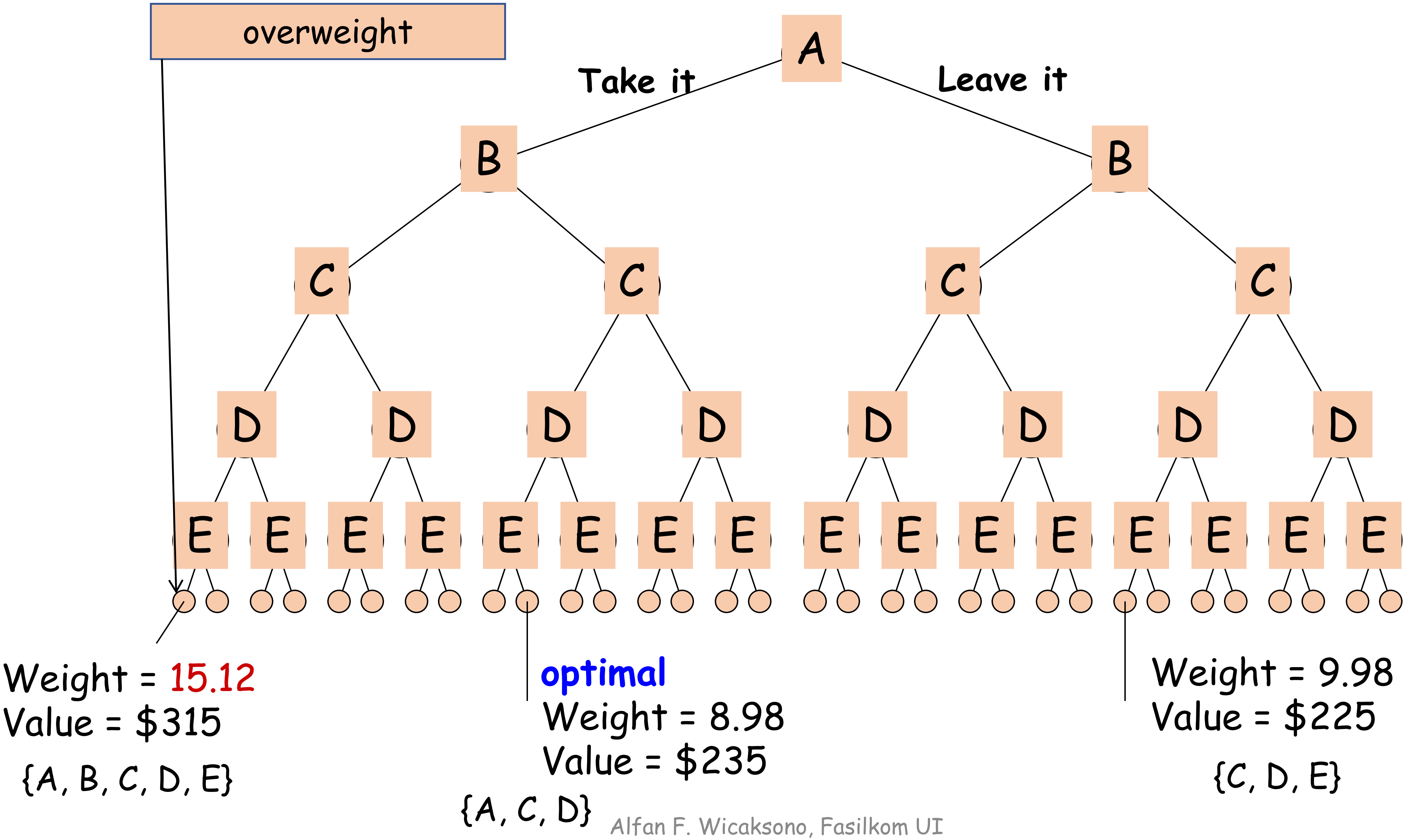
Item	Value	Weight
A	40	2
B	50	**π (3.14)**
C	100	**1.98**
D	95	5
E	30	3

State Space Tree for A Problem

- Given a problem, a **state space** is the set of all possible candidate solutions to the problem.

- We can create the candidate solutions by constructing a **state space tree**.
  - **Nodes**: partial solutions
  - **Edges**: choices in expanding partial solutions

State Space Tree for 0-1 Knapsack



Brute-Force Solution

- We examine all possible solutions;
- For 0-1 Knapsack problem with **n** items, there are 2^n solutions to be generated;

- In general, we then search the entire state space search.
  - **Depth-First Search (DFS)**

    Traverse "deeper" whenever possible & Similar to pre-order tree traversals.
  - **Breadth-First Search (BFS)**

    Traverse "wider" whenever possible & Similar to level-by-level tree traversals

Brute-Force Solution: **Depth First Tree Search**

```
dfs_tree_search(node v):
    visit(v)
    for c ∈ child(v):
        dfs_tree_search(c)
```

```
dfs_tree_search_iter(node v):
    s = Stack({})
    s.push(v)
    while not s.empty():
        e = s.pop()
        visit(c)
        for c ∈ child(e):
            s.push(c)
```

For 0-1 Knapsack Problem, the total number of nodes in the state space tree for **n** items is $2^{n+1} - 1 = O(2^n)$

Do we still want to improve?

- Can we somehow improve the brute-force solution via Depth First Tree Search?

- **Backtracking**
  - If we reach a point where a solution no longer is feasible, there is no need to continue exploring; we then **backtrack** from this point!

- **Branch-and-Bound** (for optimization problems)
  - We can backtrack if we know the best possible solution in current subtree is worst than current best solution obtained so far.

Backtracking

- **Backtracking** is a systematic way to go through a search space by traversing the state space using a depth-first search **with pruning, i.e.,** by **cutting down some "non-promising" branches**.
- Backtracking gives a significant advantage over an exhaustive brute-force search of the state space tree for the average problem.

- Elements of backtracking:
  - Performing a DFS of a state space tree;
  - Checking whether each node is **promising**, i.e., whether there is a potential that a solution might be found.
  - **Backtracking to the node's parent if it is non-promising**.

Backtracking for Finding a Solution

```
backtrack_rec(node v):
    visit(v)
    if promising(v) then
        if is_solution(v) then
            print(v)
        else
            for c ∈ child(v):
                backtrack_rec(c)
```

```
backtrack_iter(node v):
    s = Stack({})
    s.push(v)
    while not s.empty():
        e = s.pop()
        visit(e)
        if promising(e) then
            if is_solution(e) then
                print(e)
            else
                for c ∈ child(e):
                    s.push(c)
```

Exercise: Modify the two codes so that they can be used for

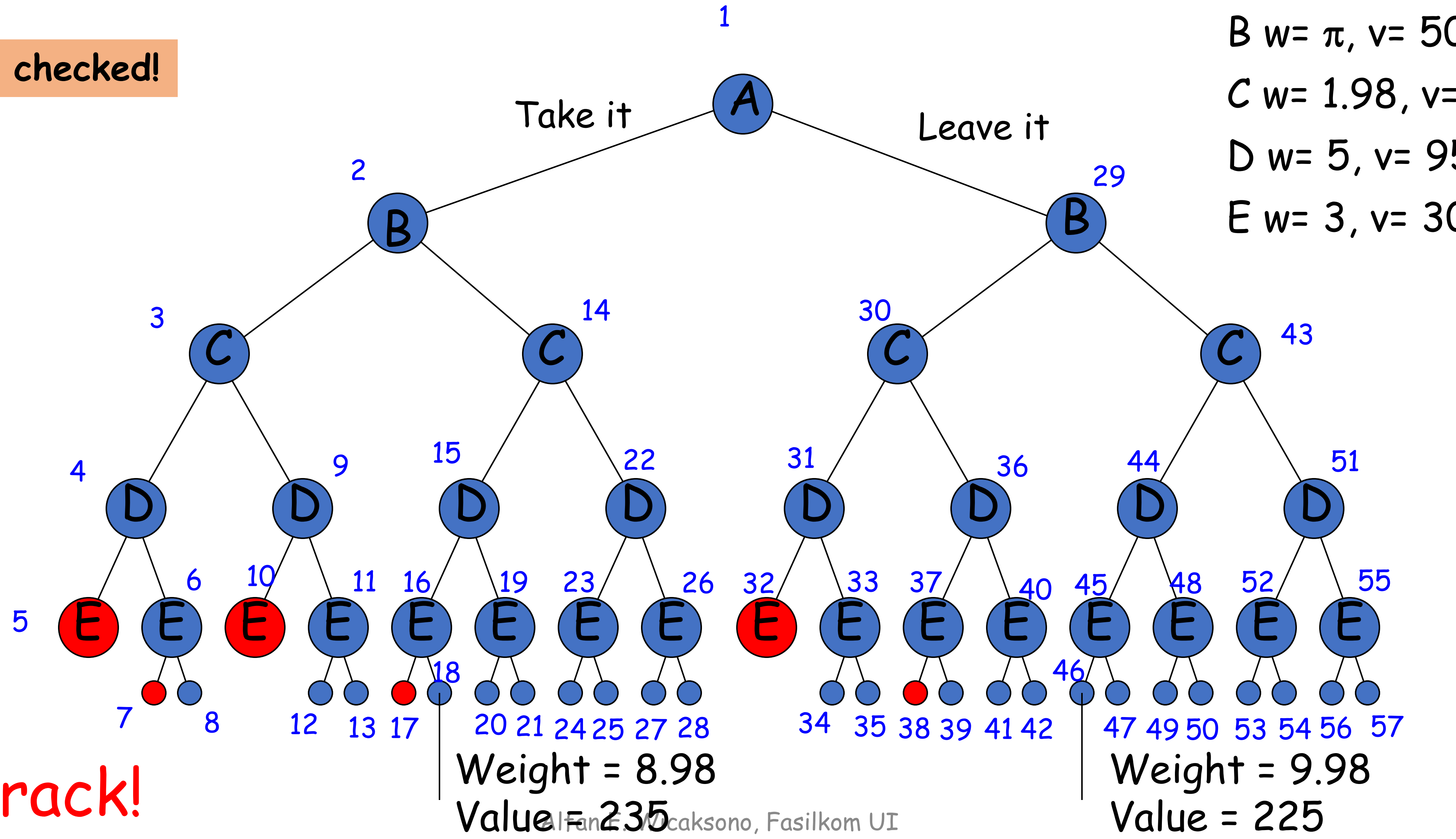
- Finding all solutions
- Finding the number of solutions

Backtracking for Optimization Problems

```
backtrack_iter(node v):
    s = Stack({})
    best = -INF
    s.push(v)
    while not s.empty():
        e = s.pop()
        visit(e)
        if promising(e) then
            best = max( best, value(e) )
            for c ∈ child(e):
                s.push(c)
    return best
```

Backtracking – Ex: 0-1 Knapsack Problem

57 nodes checked!



A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Backtrack!

Backtracking – Ex: 0-1 Knapsack Problem

```
def backtrack_knapsack(ws, vs, W):
    s = []                # empty stack
    n = len(ws)
    max_profit = 0   # keep tracking the profit of the best solution so far
    s.append((0, 0, W))      # state/node: (level, current profit, current capacity)
    while s != []:
        (l, v, w) = s.pop()
        if w >= 0:                        # if promising
            max_profit = max([max_profit, v])
            if l < n:                # if l == n, we stop since we have inspected last item
                leave_i = (l+1, v, w)
                take_i = (l+1, v + vs[l], w - ws[l])
                s.append(leave_i)       # push "leave item i" child node
                s.append(take_i)        # push "take item i" child node
    return max_profit
```

Exercise!

- Suppose we have two buckets, A & B. **A** has a capacity of **5 liters**, and **B** has a capacity of **3 liters**.
- You have several possible actions:
  - You can fill bucket A until bucket A is full, or vice versa;
  - You can pour bucket A into bucket B until bucket B is full, or vice versa;
  - …
- Can we find a way to get **4 liters of water**?

- Devise a possible state space, and an algorithm to answer the question.

Exercise!

- We still consider the previous problem.
- Now, we want to optimize the number of steps; we want a solution with a minimum number of steps.

- Can you modify your previous solution?

Can we still improve the previous solution? **Branch-and-Bound**!

- **Branch-and-Bound** : We can backtrack if we know the best possible solution in current subtree is worst than current best solution obtained so far.

- A node has a **bound**: an upper bound on the profit we could achieve by expanding beyond the node!

Can we still improve the previous solution? **Branch-and-Bound!**

- A node is **non-promising** if:
  - The **total weight >= W**
  - The **maxprofit** so far **>= bound** (potential upper bound of profit)

- **Bound**: An estimate for improvement (pretending fractional knapsack) given **current ordering (A -> B -> C -> D -> E)**. For example,
  - A down (**take A and all after A**) give \$244,72
    - 40 + 50 + 100 + (2,88/5) * 95 = 244,72
  - B down give 50 + 100 + (4,88/5)*95 = \$242,72
    - We leave A
  - C down give \$225
    - We leave A and B
  - D down give \$125
    - We leave A, B, and C
  - ...

W = 10

A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Branch-and-Bound – Ex: 0-1 Knapsack Problem

$W = 10$

A w= 2, v= 40
B w= π, v= 50
C w= 1.98, v= 100
D w= 5, v= 95
E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 0

(0, 10, 244.72)

A

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 40

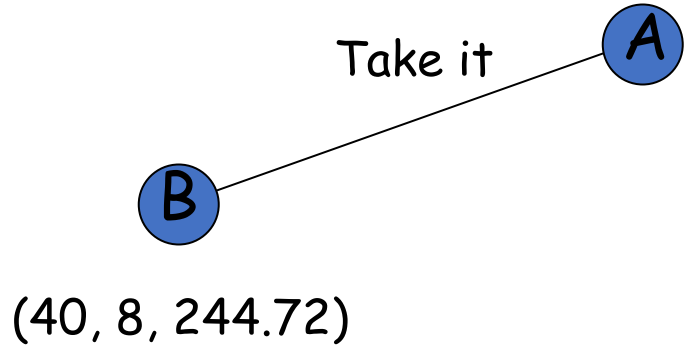
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



Upper bound = when we take A, and we take B and all remaining items
= 244.72

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

A w= 2, v= 40

B w= π, v= 50

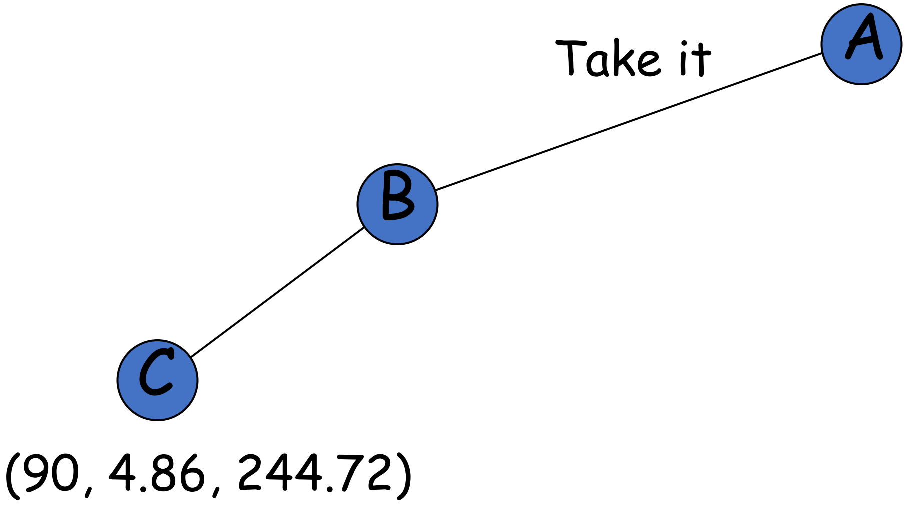
C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 90



Upper bound = when we take A & B, and we take C and all remaining items
= 244.72

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

A w= 2, v= 40

B w= π, v= 50

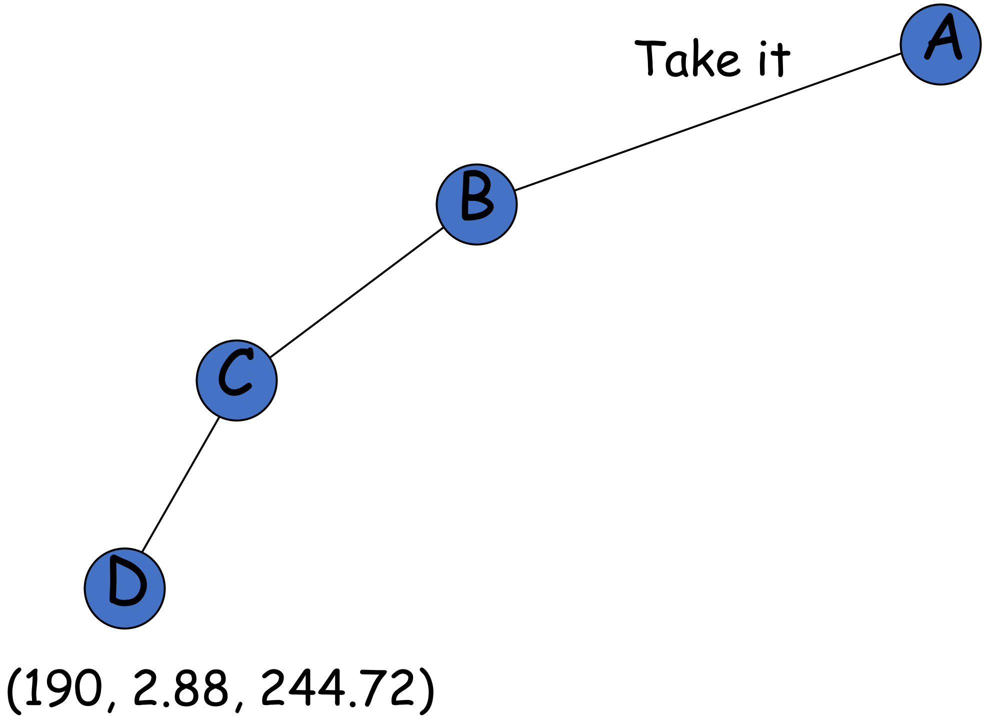
C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190



Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190

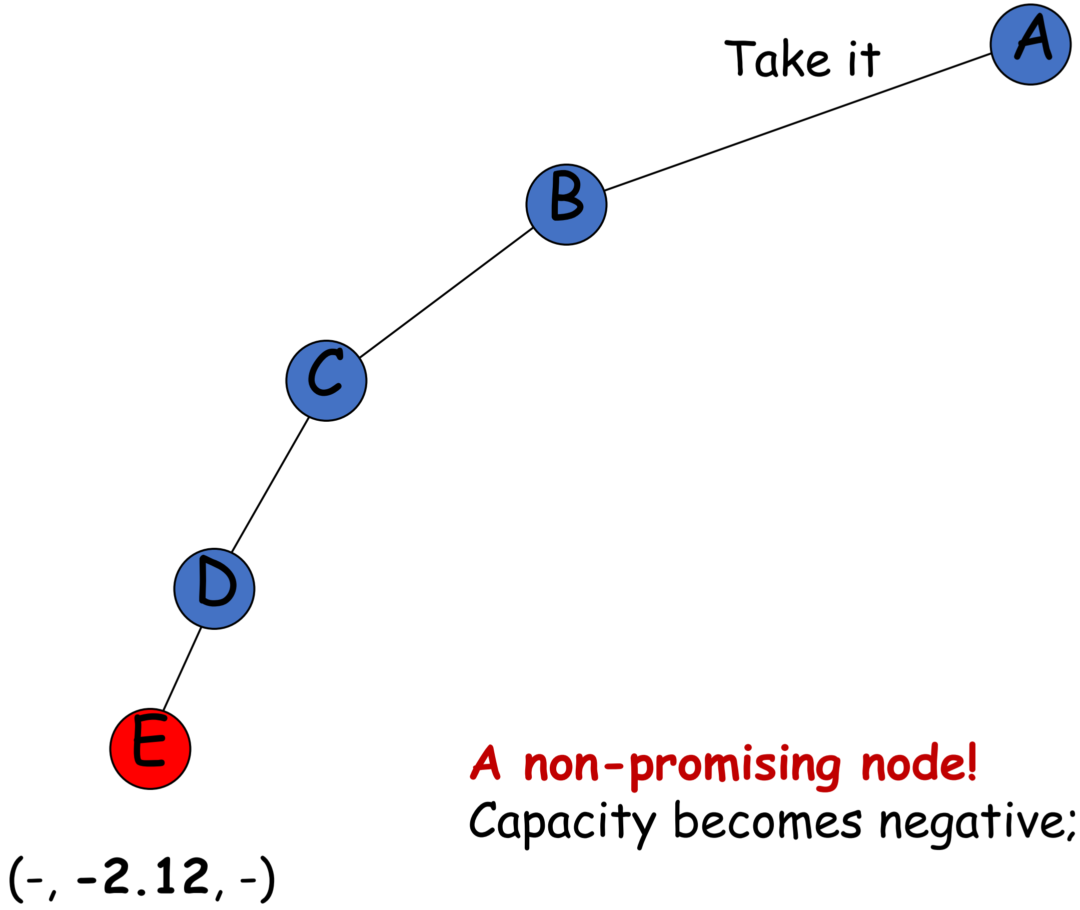
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!
Capacity becomes negative; total weight > W

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190

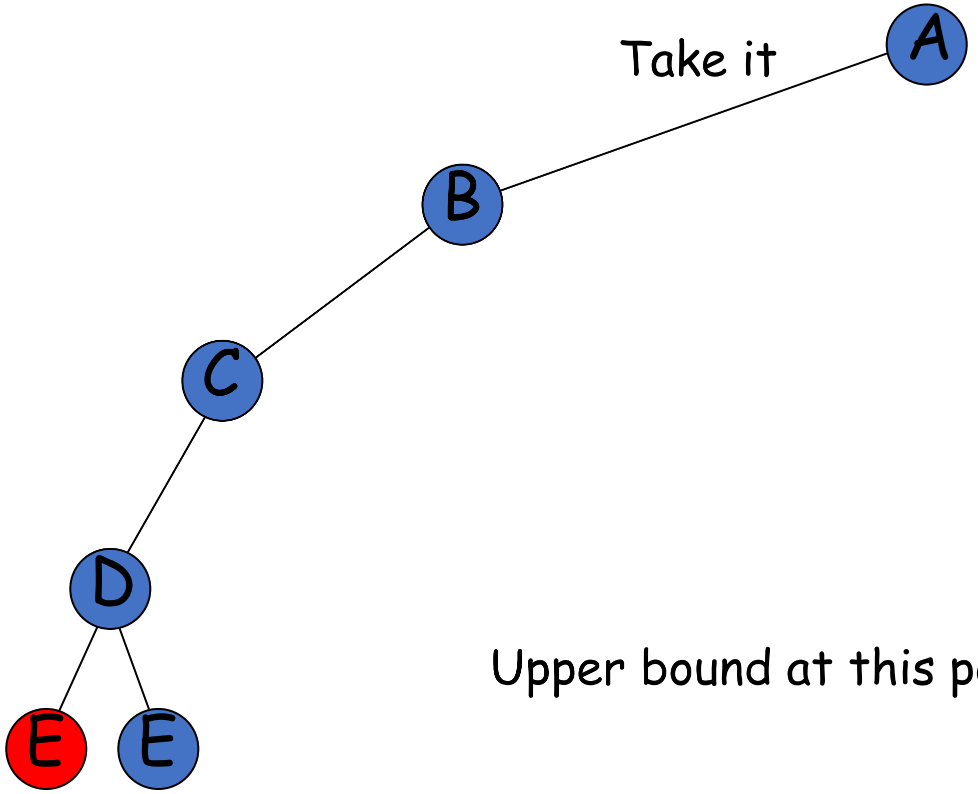
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



(190, 2.88, 218.8)

Upper bound at this point = 190 + (2.88/3)*30 = 218,8

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190

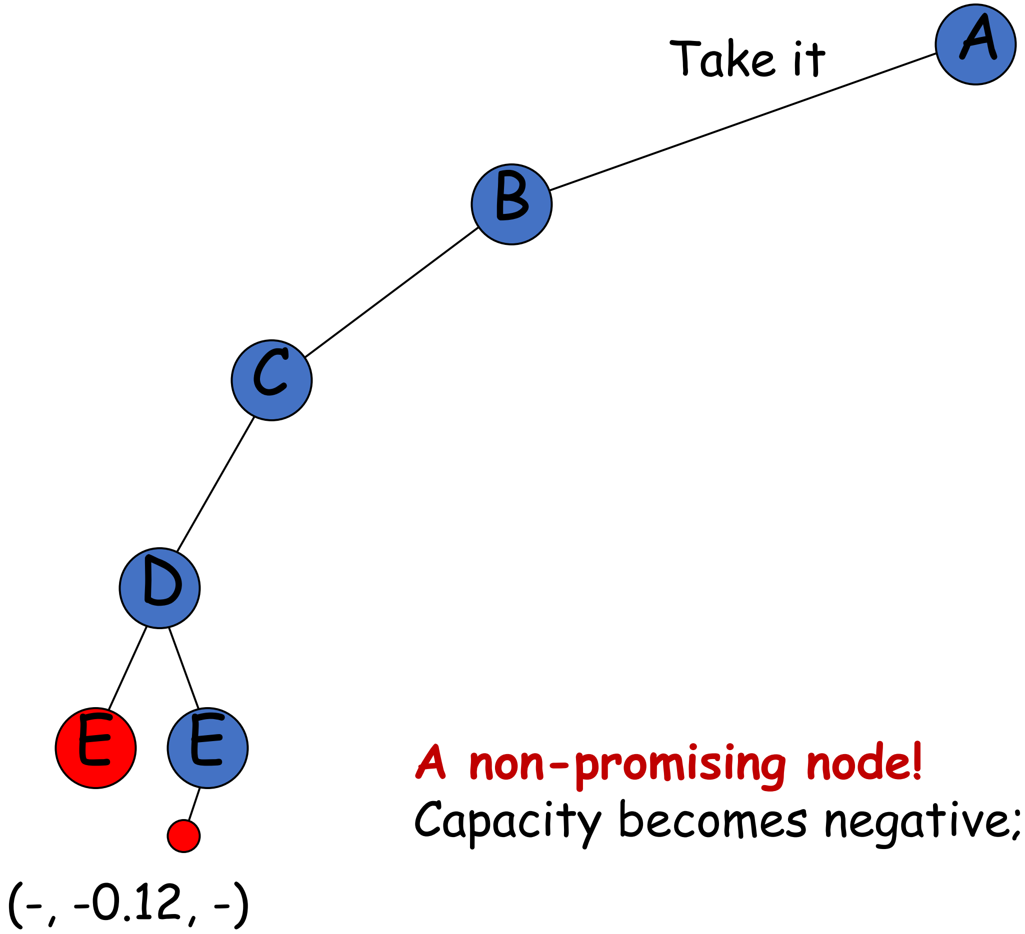
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!
Capacity becomes negative; total weight > W

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190

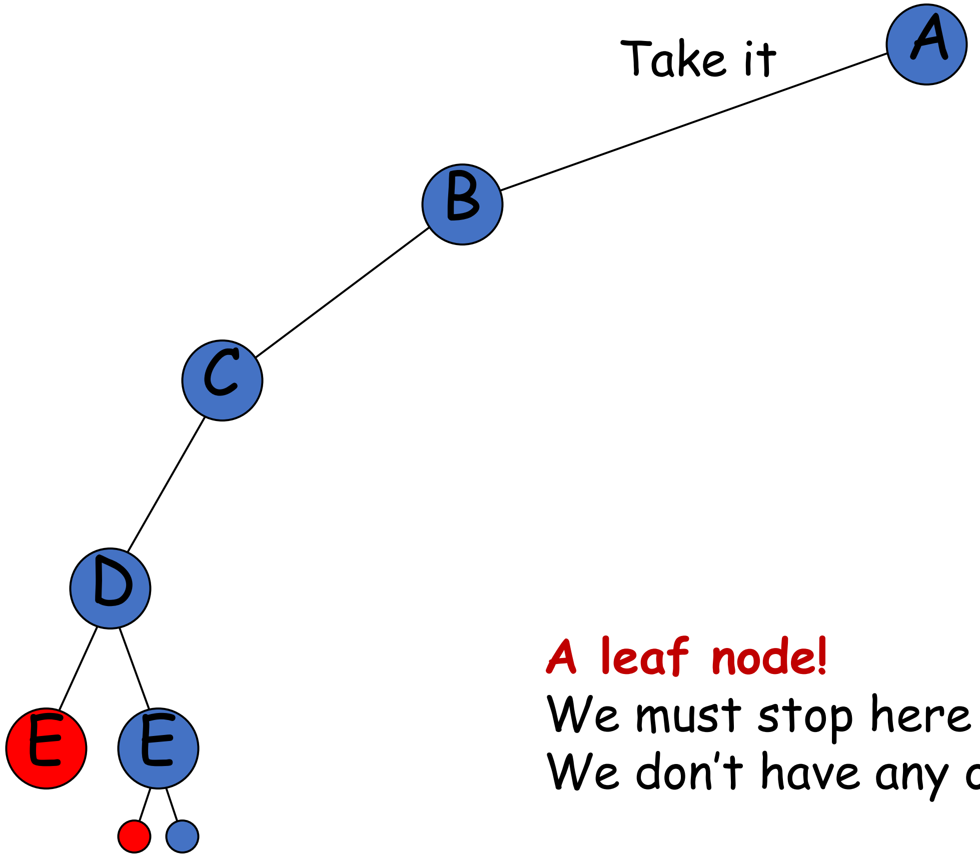
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



(190, 2.88, 190)

A leaf node!
We must stop here since we have inspected the last item E.
We don't have any other item beyond E.

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

A w= 2, v= 40

B w= π, v= 50

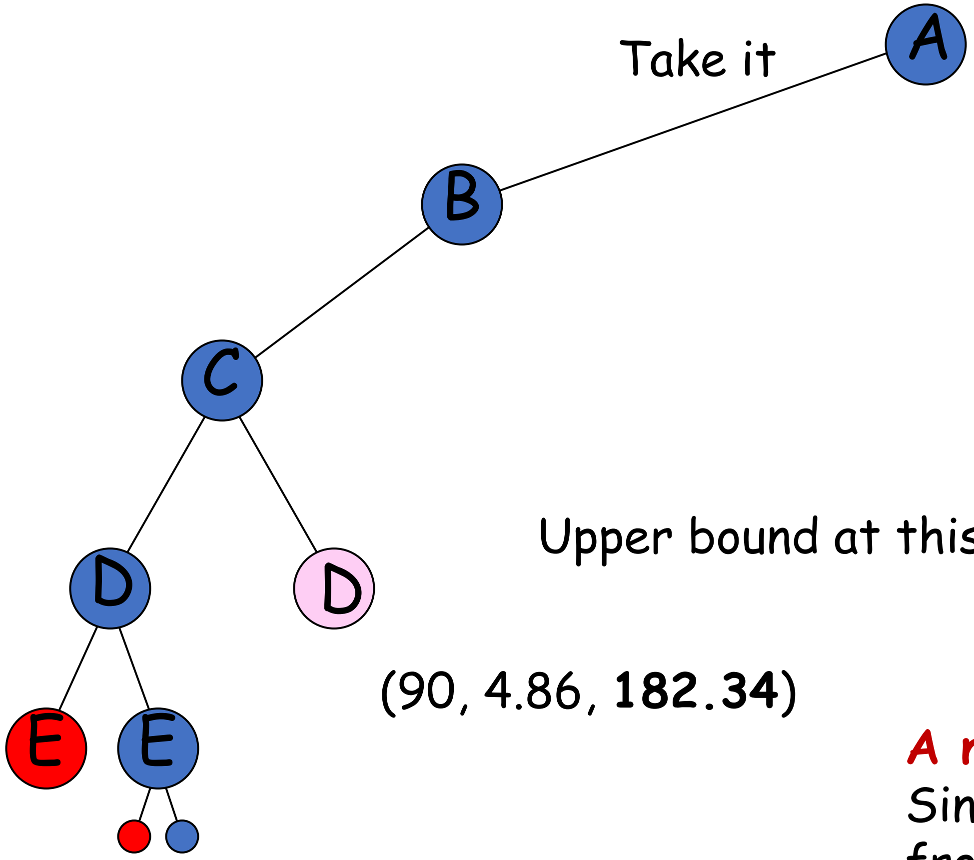
C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190



Upper bound at this point = 90 + (4.86/5)*95 = 182,34

(90, 4.86, **182.34**)

A non-promising node!
Since the upper bound is 182.34, if we continue from this node, we'll never reach any solution with total value > 190.

Branch-and-Bound – Ex: 0-1 Knapsack Problem   W = 10

A w= 2, v= 40

B w= π, v= 50

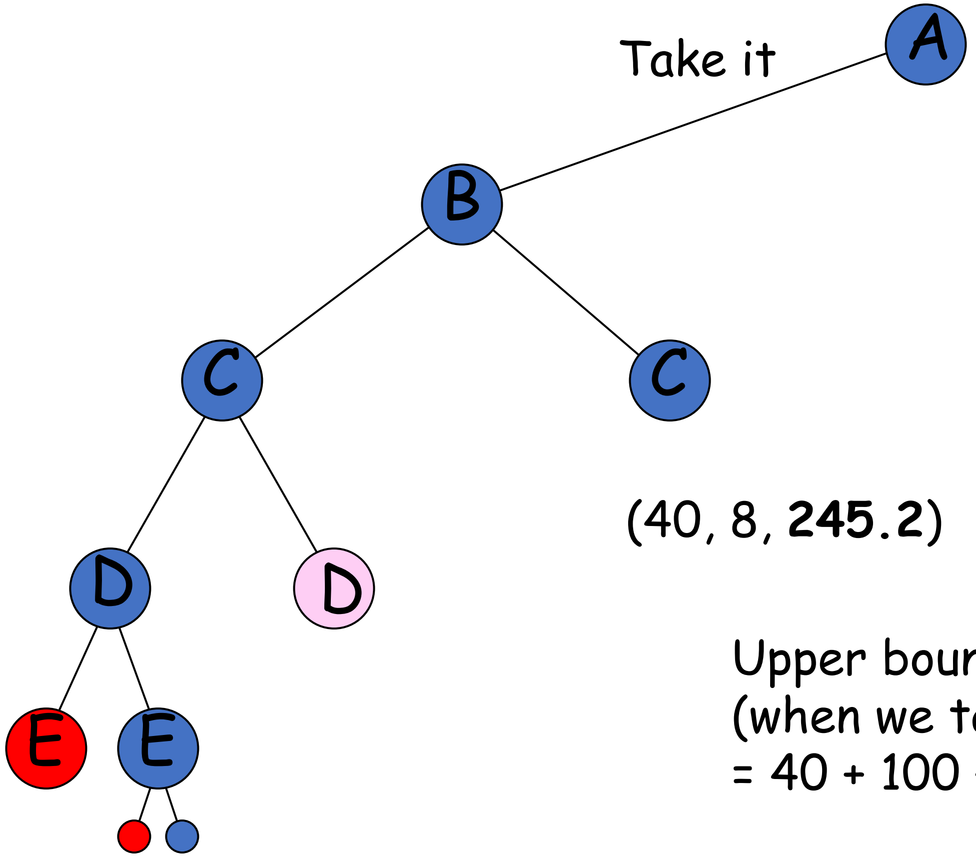
C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190



(40, 8, **245.2**)

Upper bound at this point
(when we take A, leave B, and **take C, D & E**)
= 40 + 100 + 95 + (1.02/3)*30 = 245.2

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 190

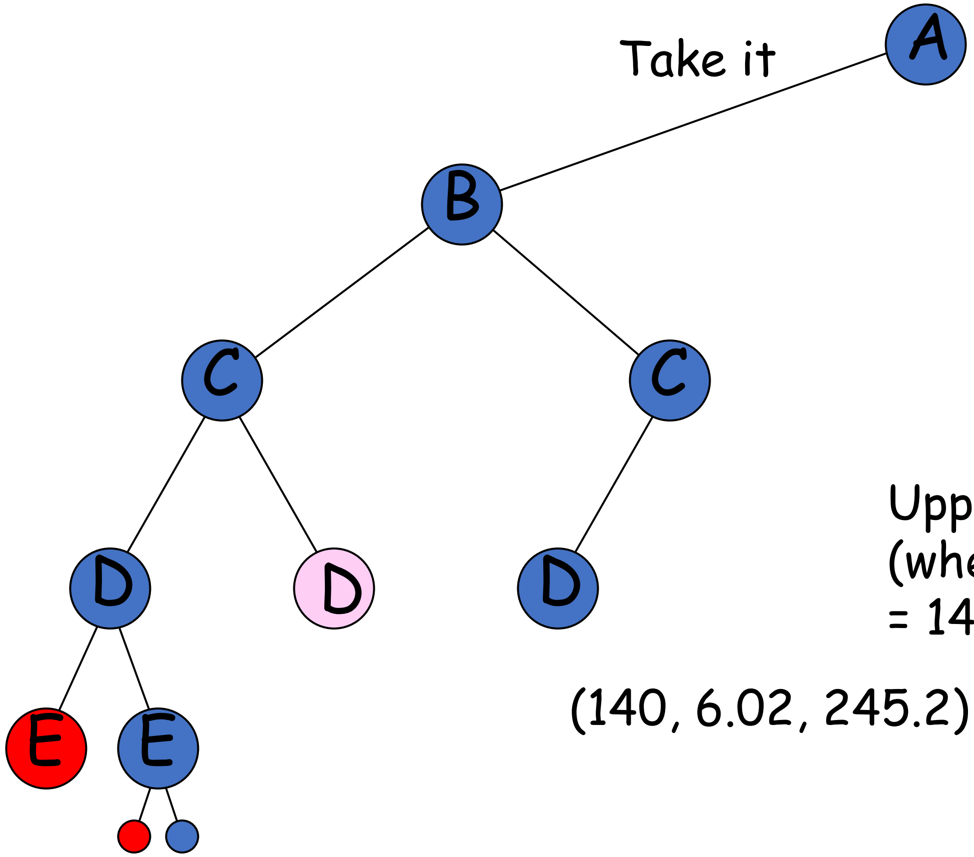
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



Upper bound at this point
(when we take A, leave B, take C, **and take D & E**)
= 140 + 95 + (1.02/3)*30 = 245.2

(140, 6.02, 245.2)

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

A w= 2, v= 40

B w= π, v= 50

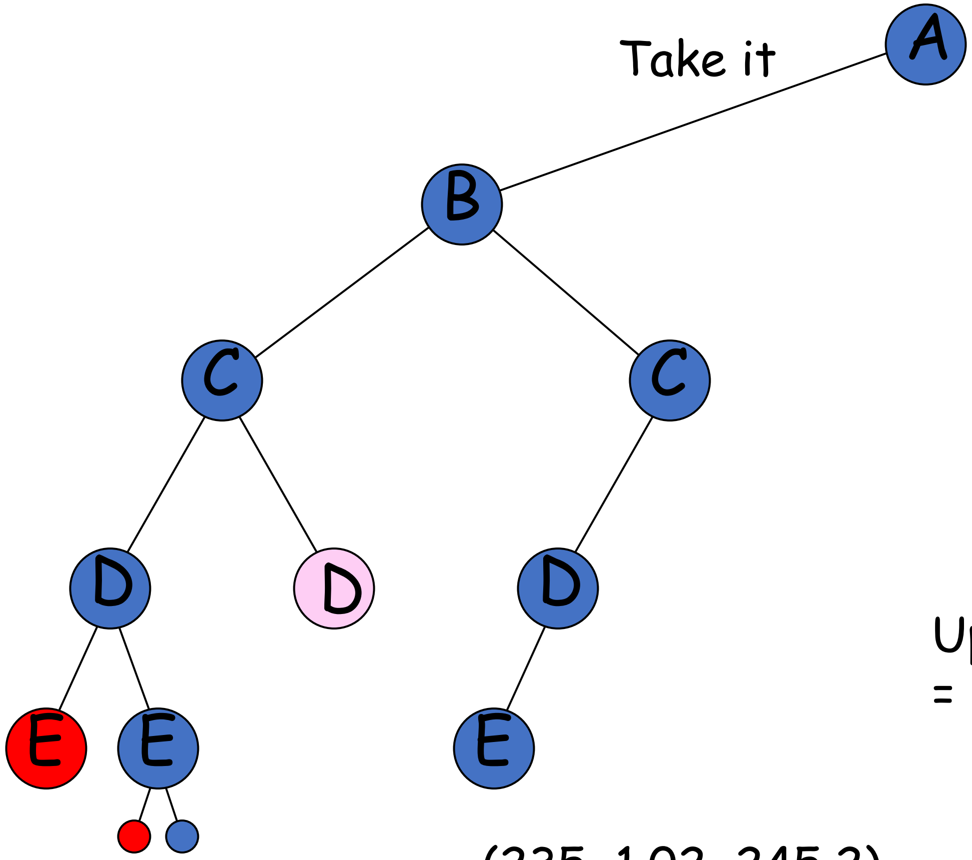
C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235



Upper bound at this point
= 235 + (1.02/3)*30 = 245.2

(235, 1.02, 245.2)

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

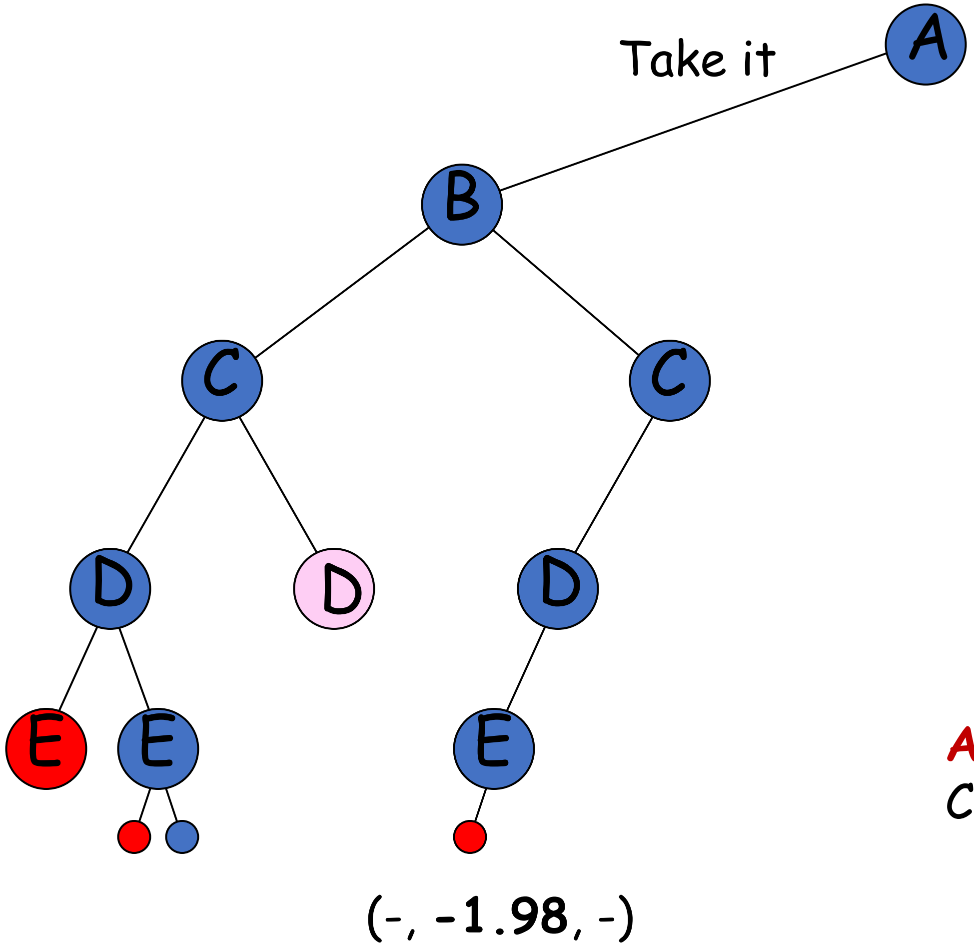
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!

Capacity becomes negative; total weight > W

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

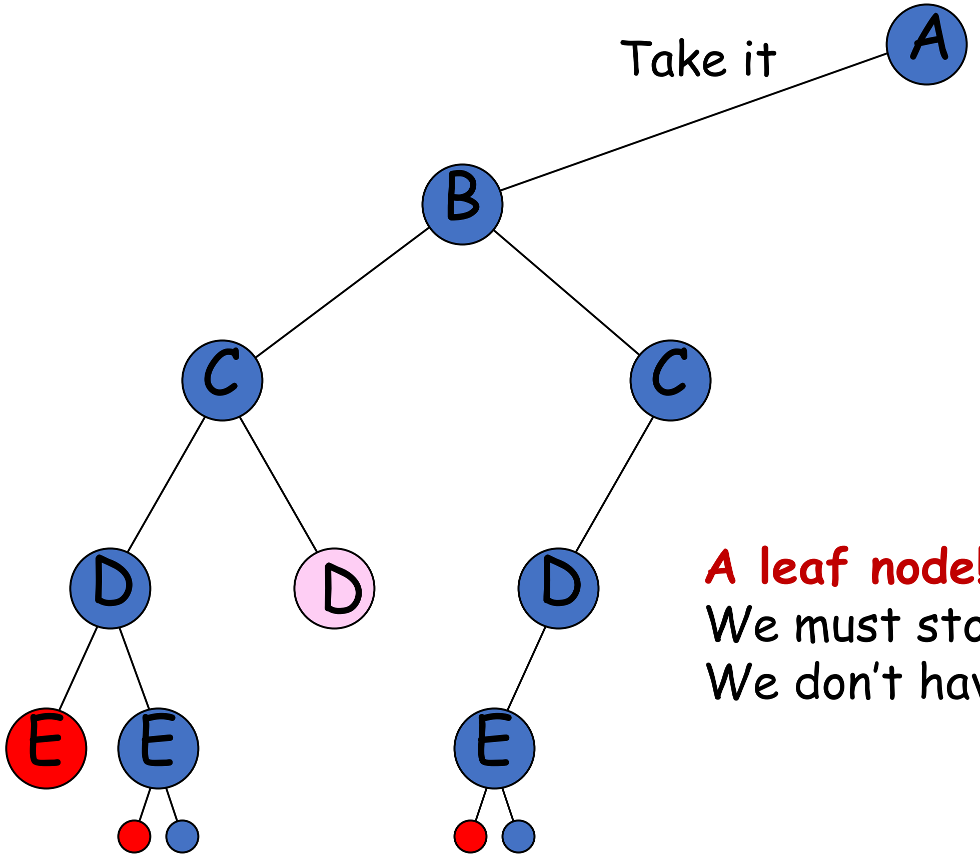
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A leaf node!
We must stop here since we have inspected the last item E.
We don't have any other item beyond E.

(235, 1.02, 245.2)

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

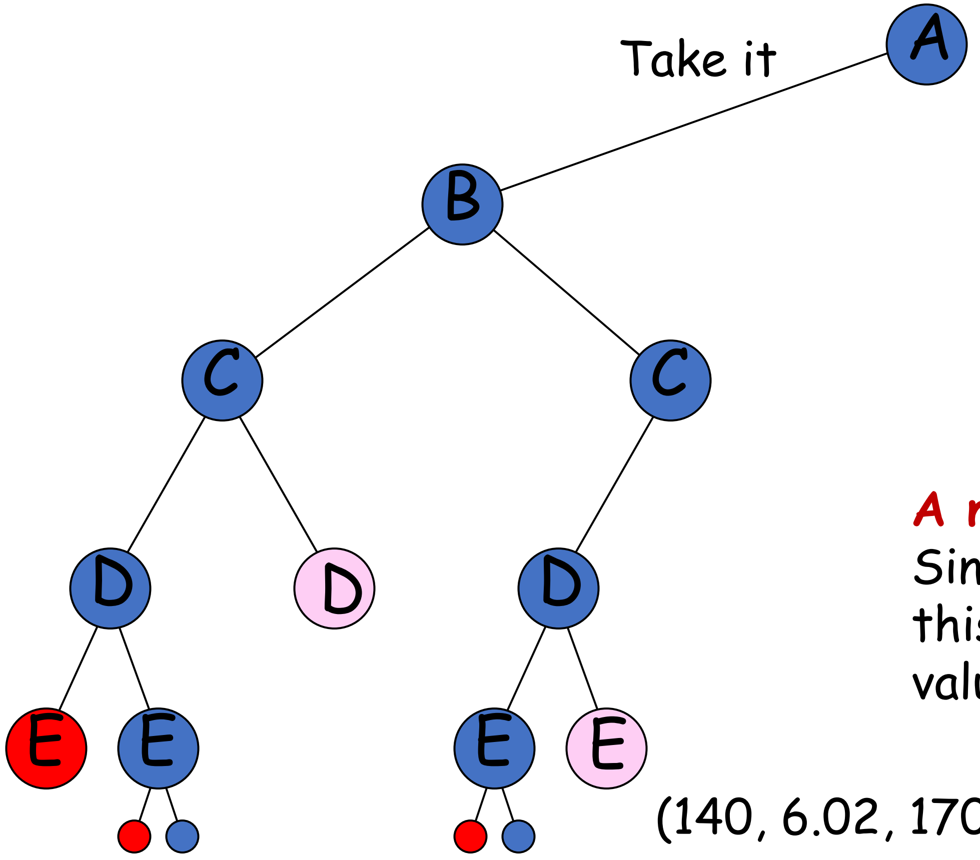
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!
Since the upper bound is 170, if we continue from this node, we'll never reach any solution with total value > 235.

(140, 6.02, 170)

Upper bound at this point
= 140 + 30 = 170

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

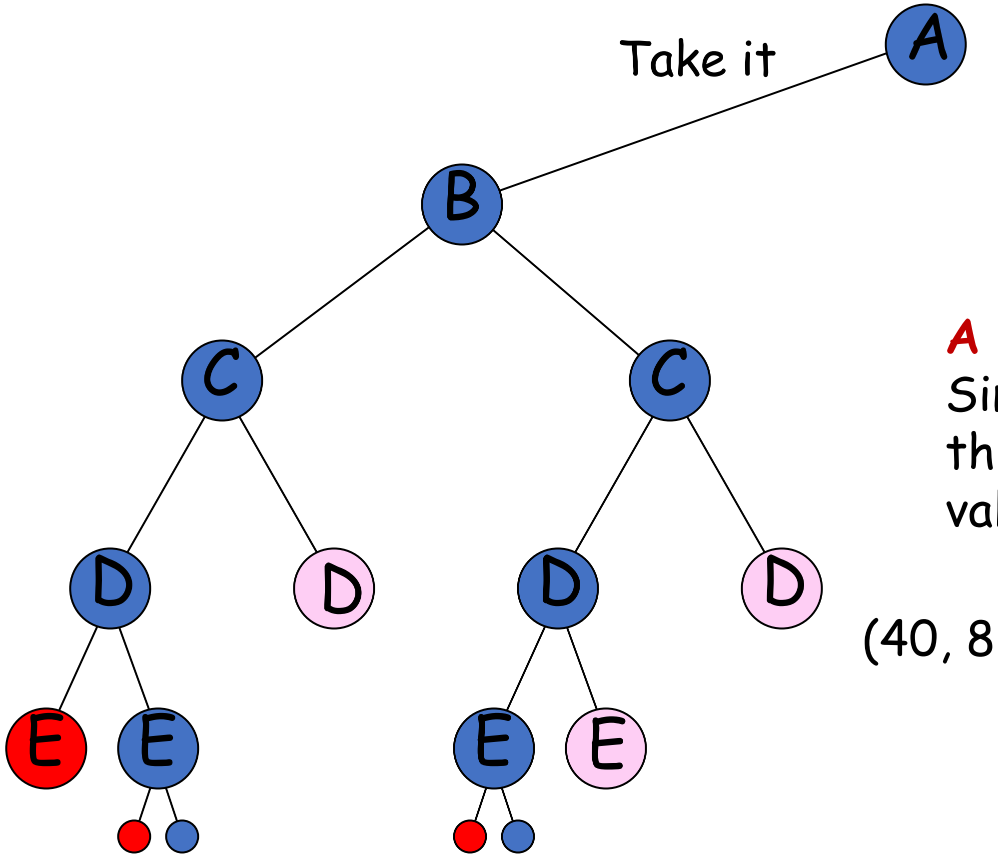
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



(40, 8, 165)

A non-promising node!
Since the upper bound is 165, if we continue from this node, we'll never reach any solution with total value > 235.

Upper bound at this point
= 40 + 95 + 30 = 165

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

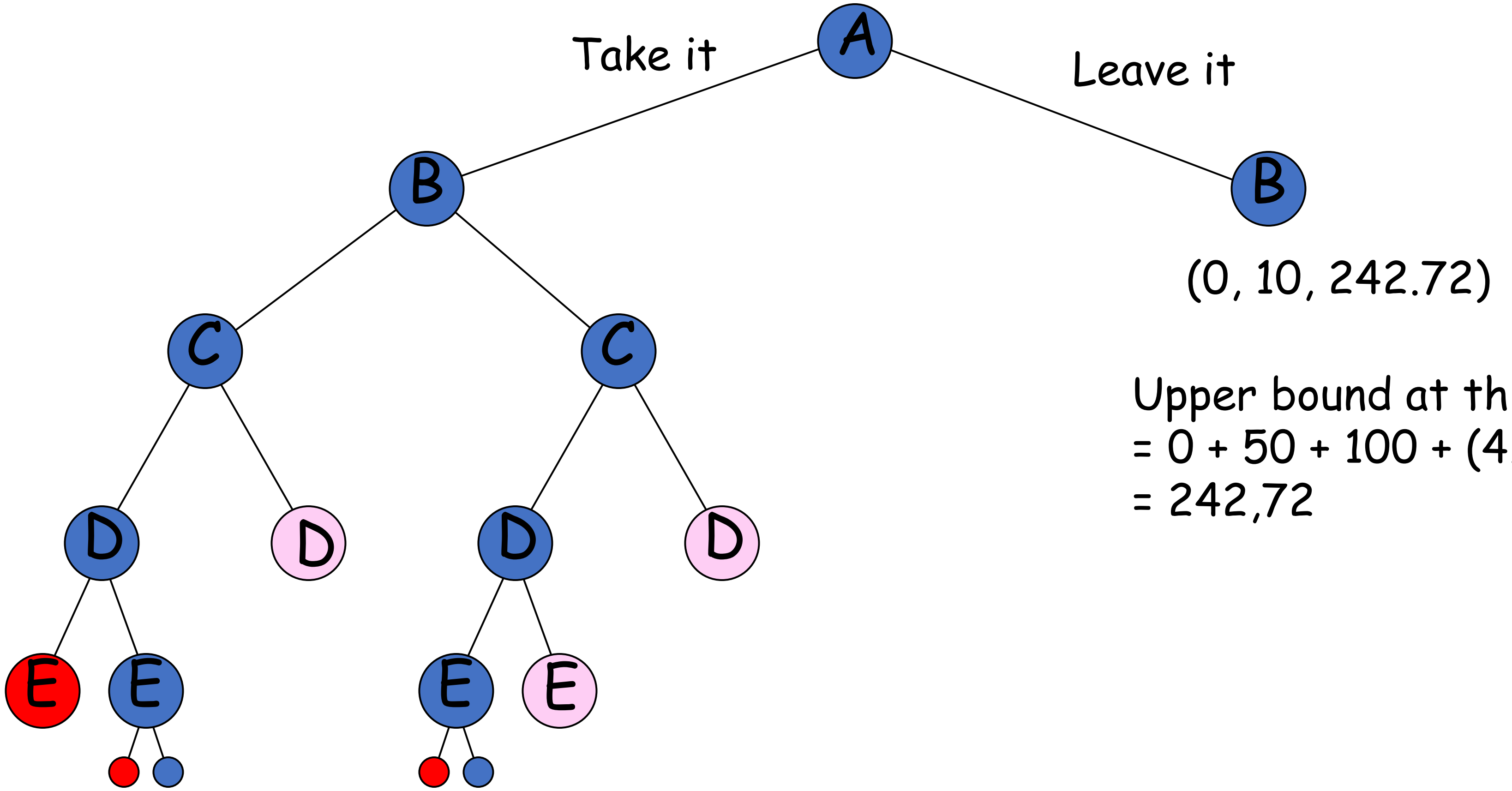
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



Upper bound at this point
= 0 + 50 + 100 + (4.88/5)*95
= 242,72

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

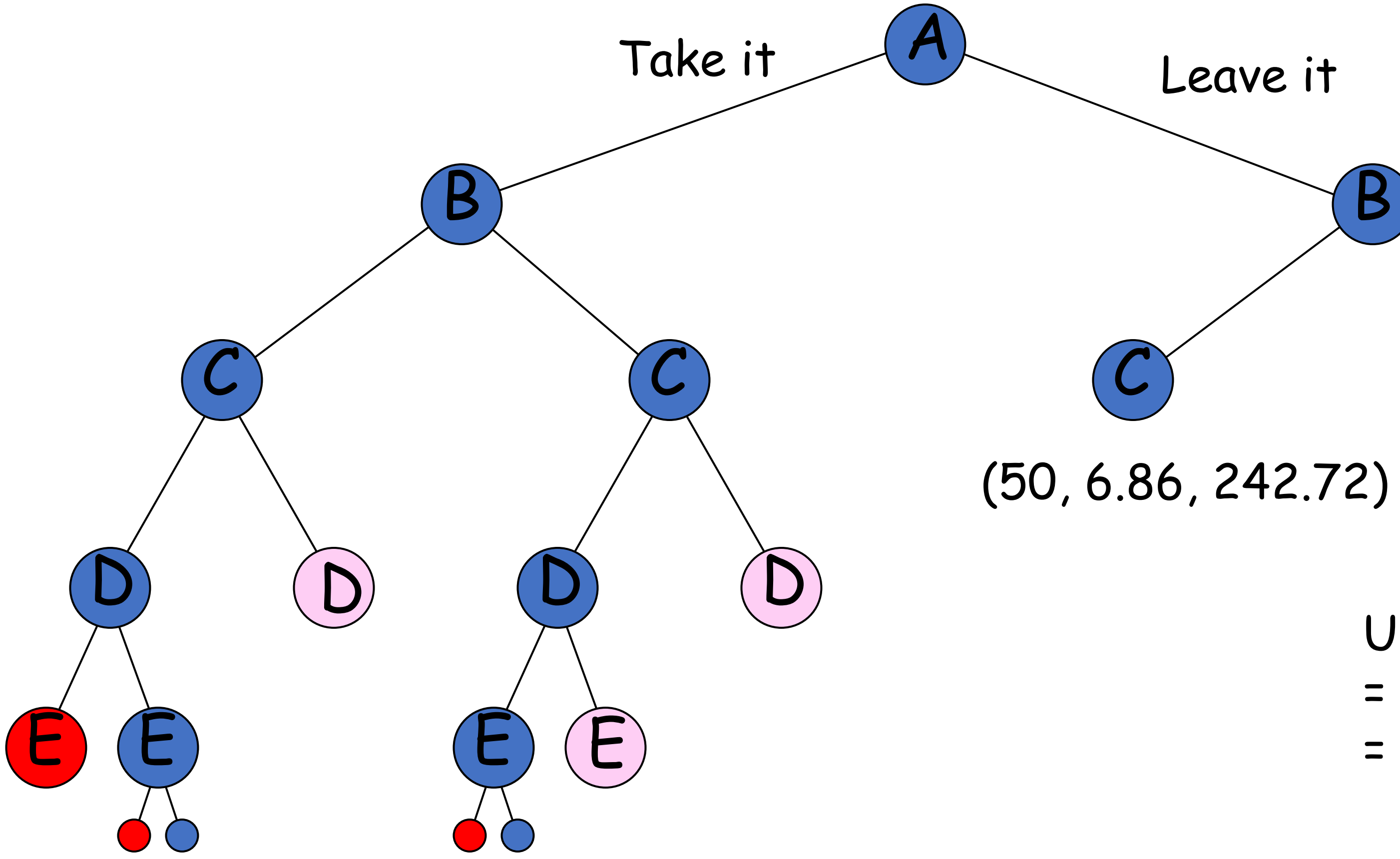
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



Upper bound at this point
= 50 + 100 + (4.88/5)*95
= 242,72

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

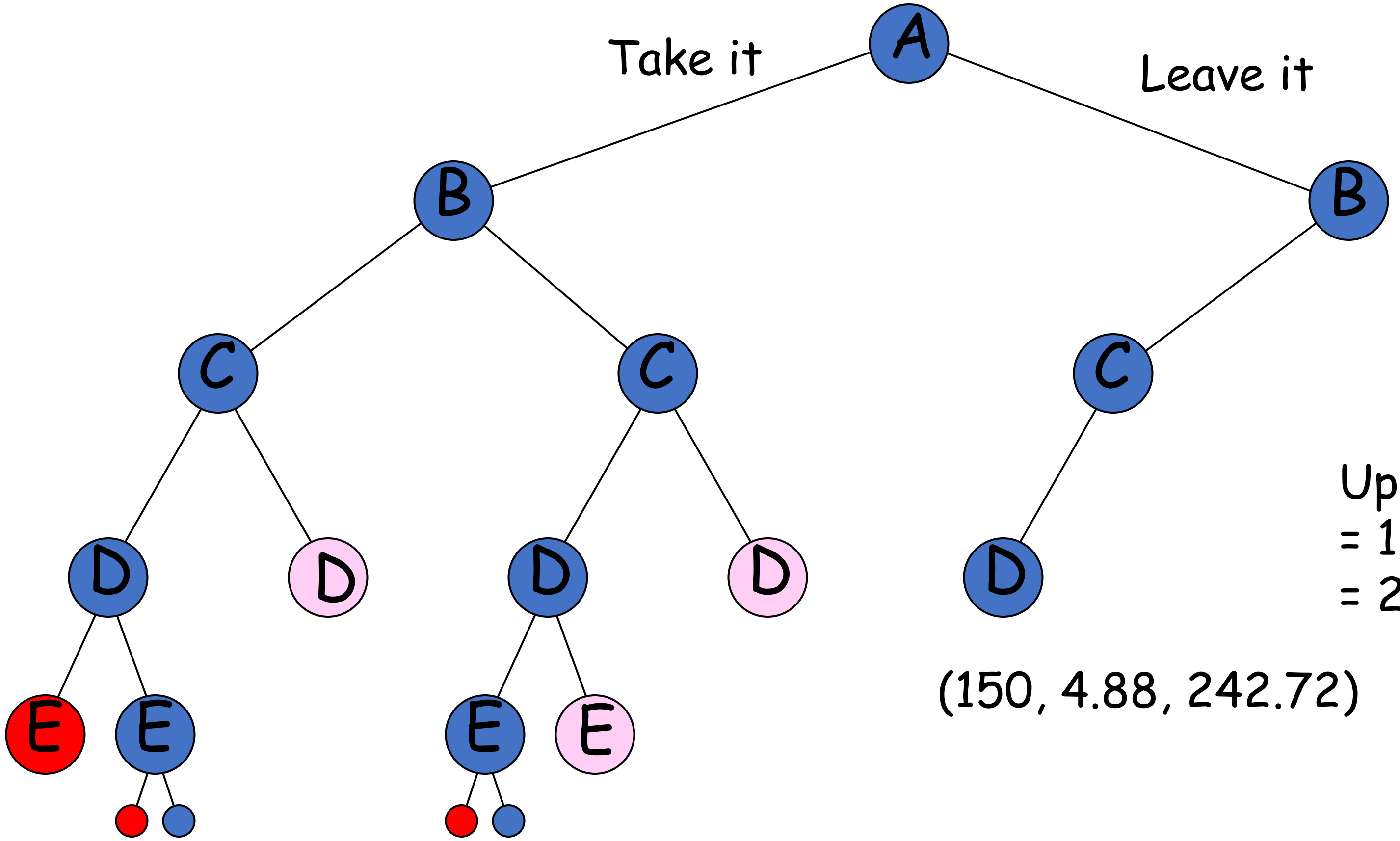
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



Upper bound at this point
= 150 + (4.88/5)*95
= 242,72

(150, 4.88, 242.72)

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

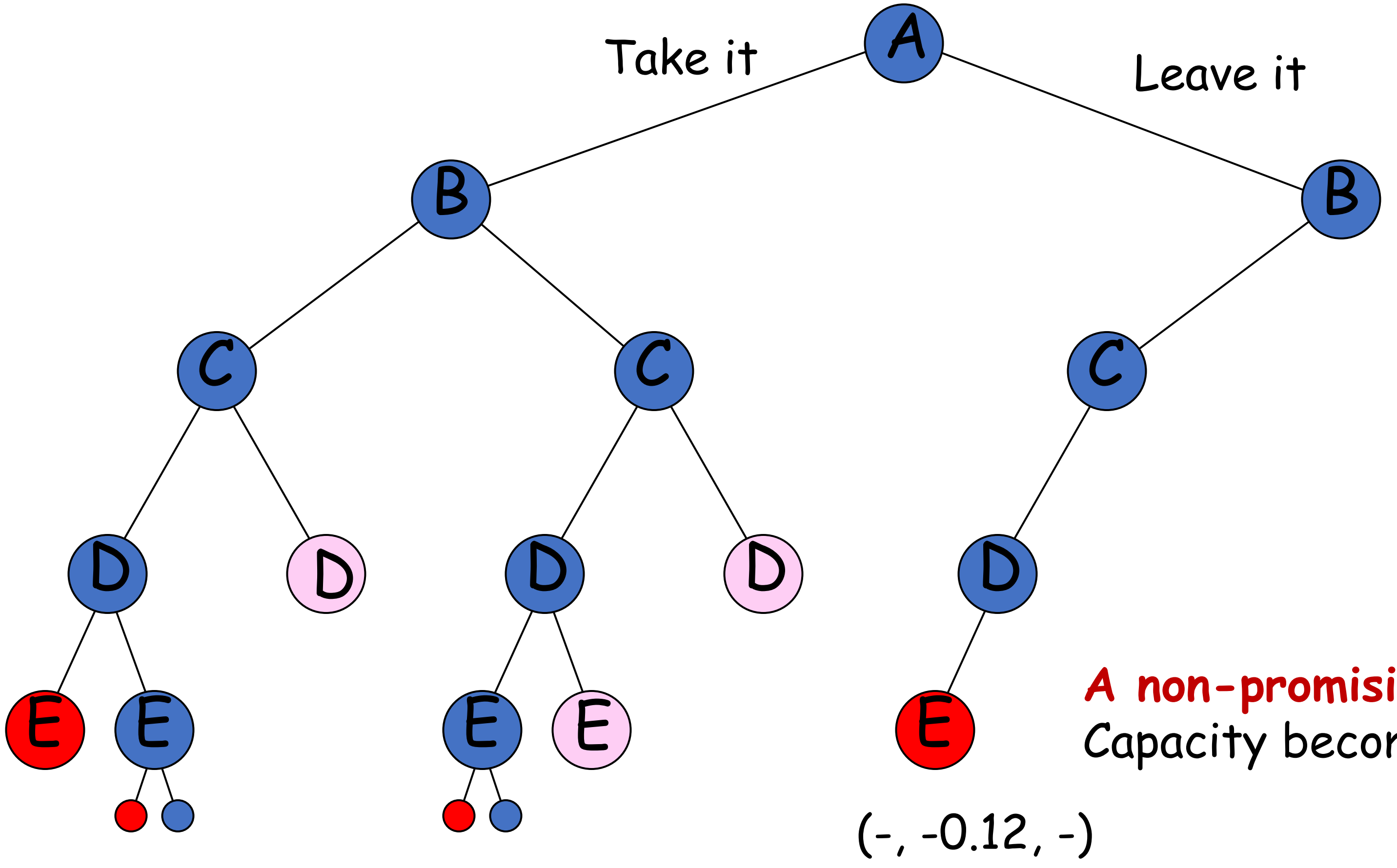
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!

Capacity becomes negative; total weight > W

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

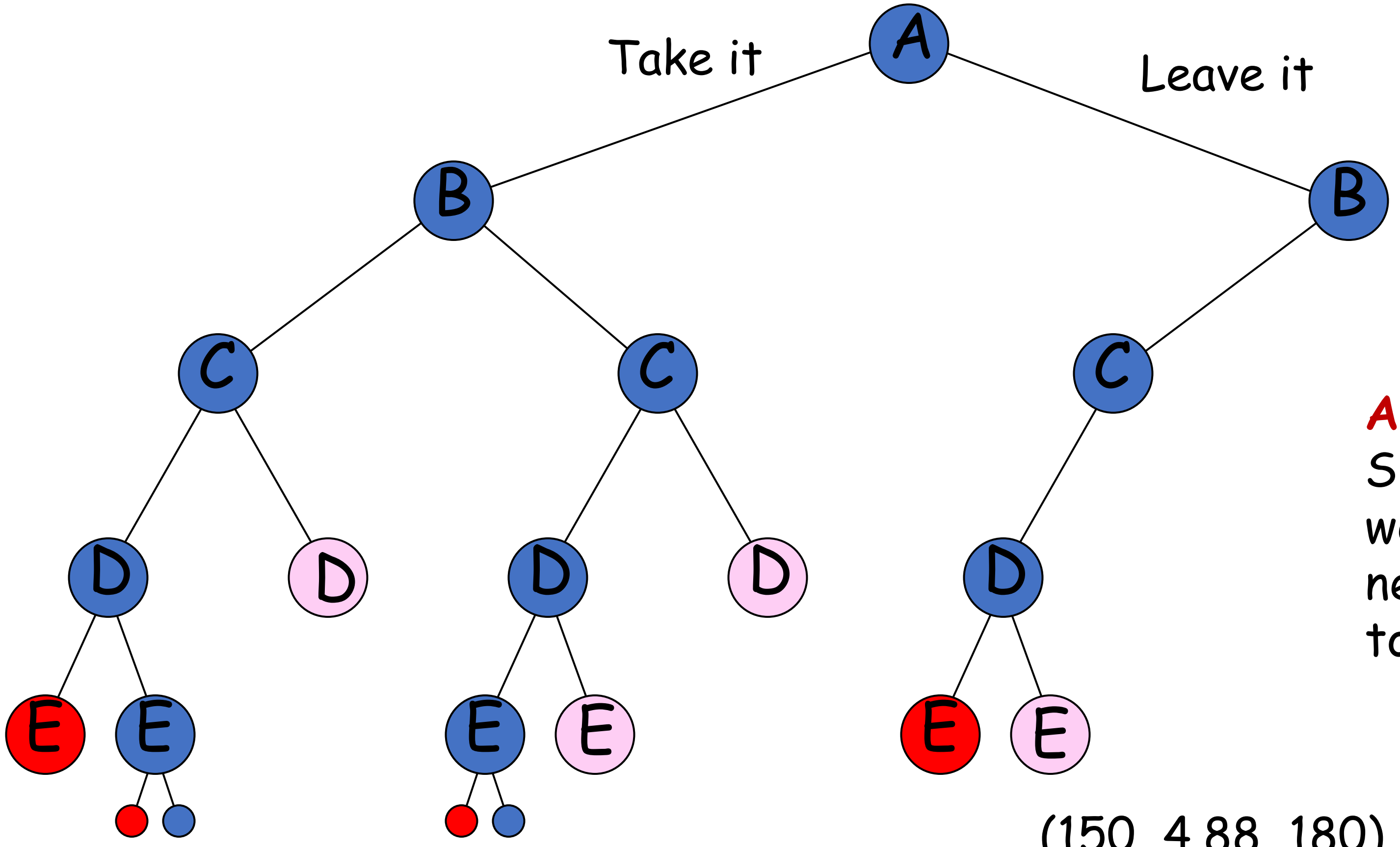
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



A non-promising node!
Since the upper bound is 180, if we continue from this node, we'll never reach any solution with total value > 235.

(150, 4.88, 180)

Upper bound at this point
= 150 + 30
= 180

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

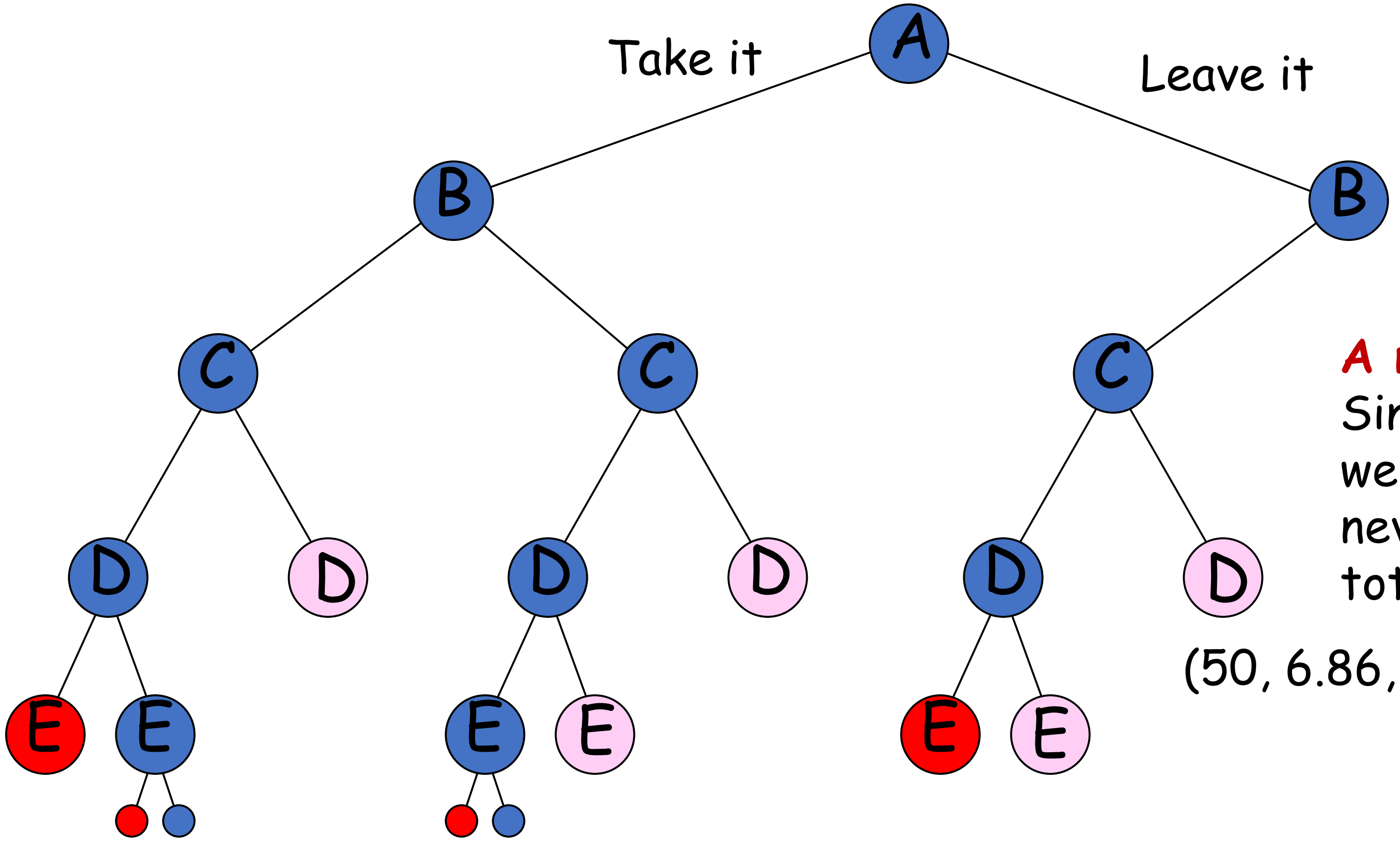
A w= 2, v= 40

B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30



(50, 6.86, 163.6)

A non-promising node!
Since the upper bound is 163.6, if we continue from this node, we'll never reach any solution with total value > 235.

Upper bound at this point
= 50 + 95 + (1,86/3)*30
= 163,6

Branch-and-Bound – Ex: 0-1 Knapsack Problem

W = 10

Each node/state = (profit, weight capacity, **bound**)

Best so far = 235

A w= 2, v= 40

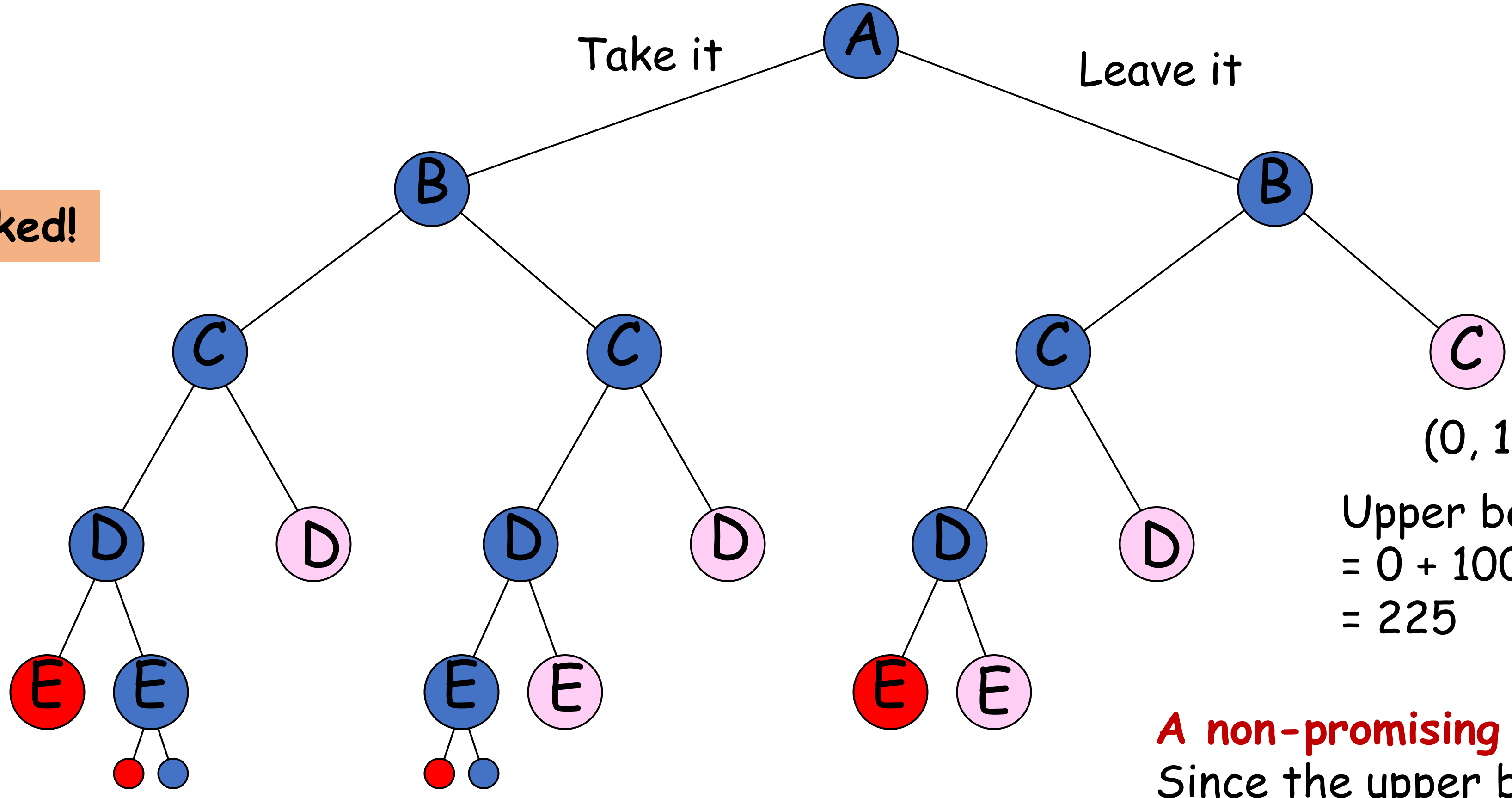
B w= π, v= 50

C w= 1.98, v= 100

D w= 5, v= 95

E w= 3, v= 30

23 nodes checked!



(0, 10, 163.6)

Upper bound at this point
= 0 + 100 + 95 + 30
= 225

A non-promising node!
Since the upper bound is 225, if we continue from this node, we'll never reach any solution with total value > 235.

Branch-and-Bound - Ex: 0-1 Knapsack Problem

```python
def bound(l, v, w, ws, vs):
    upper_bound_profit = v
    j = l

    while j < len(ws) and w >= ws[j]:
        upper_bound_profit += vs[j]
        w -= ws[j]
        j += 1

    if j < len(ws):
        upper_bound_profit += (w / ws[j]) * vs[j]

    return upper_bound_profit
```

Branch-and-Bound - Ex: 0-1 Knapsack Problem

```
def bnb_knapsack(ws, vs, W):
    s = []                  # empty stack
    n = len(ws)
    max_profit = 0   # keep tracking the profit of the best solution so far
    s.append((0, 0, W))      # state/node: (level, current profit, current capacity)
    while s != []:
        (l, v, w) = s.pop()
        if w >= 0 and bound(l, v, w, ws, vs) > max_profit:   # if promising
            max_profit = max([max_profit, v])
            if l < n:                # if l == n, we stop since we have inspected last item
                leave_i = (l+1, v, w)
                take_i = (l+1, v + vs[l], w - ws[l])
                s.append(leave_i)       # push "leave item i" child node
                s.append(take_i)        # push "take item i" child node
    return max_profit
```

A good "bound"?

- A "bound" is actually a **heuristic function**;
- The key to branch and bound is having a good initial bound;
- How might we generate a good initial bound, that is, how might we generate a good solution earlier?

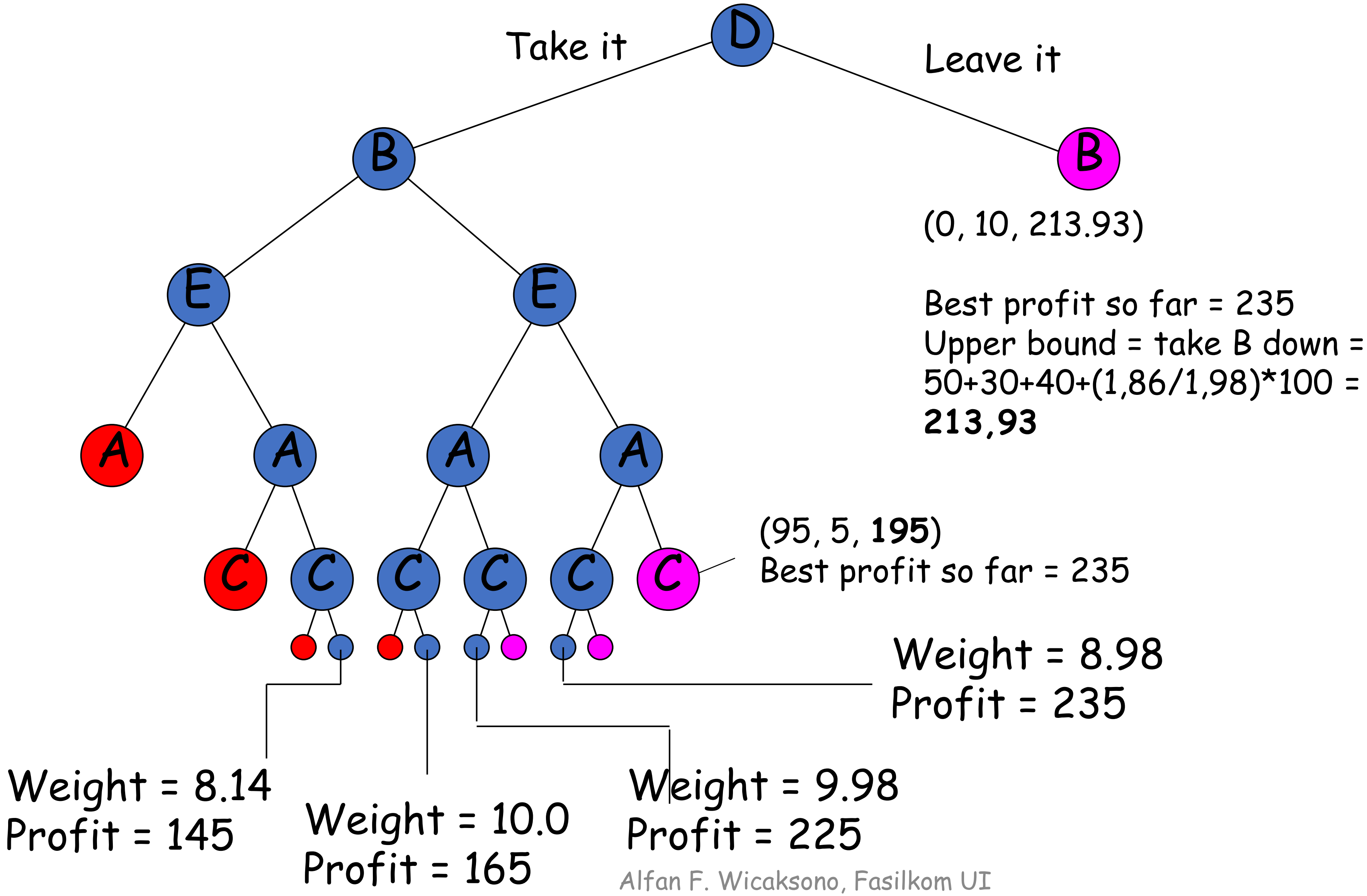
Order of Items Matters

Since there is no fixed ordering of items, is there a better way to order the input items?

- Highest weight first (Heaviest on Top)
  Generate infeasible solutions quickly
- Highest density (**profit/weight**) first
  Generate good solution quickly
- Which is better depends on input

Heaviest on Top

Items are sorted based on weights in descending order.



D: w=5, v=95

B: w= π, v= 50

E: w= 3, v= 30

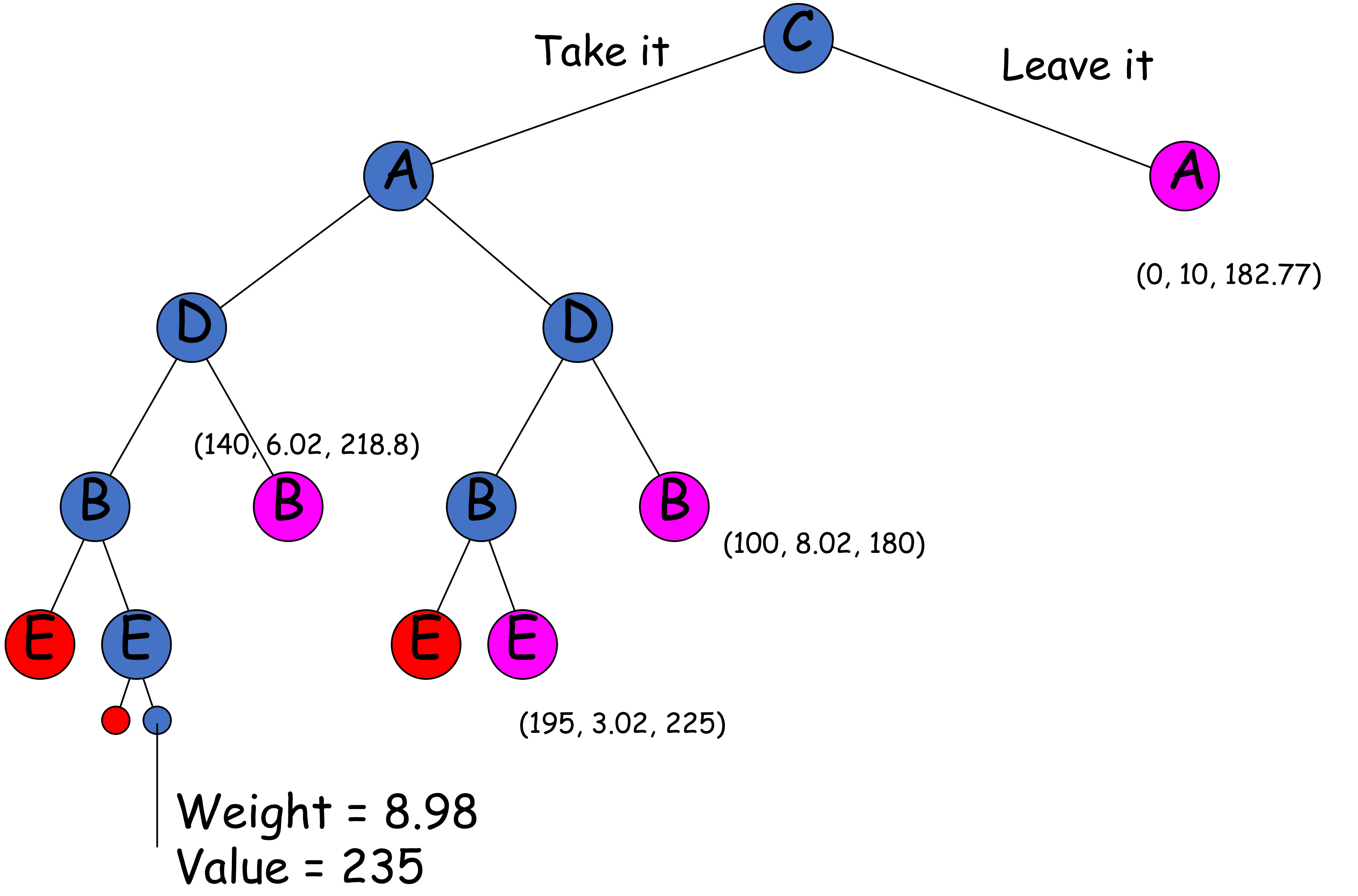
A: w= 2, v= 40

C: w= 1.98, v= 100

23 nodes checked!

Best Density on Top

Items are sorted based on **profit/weight** in descending order.



C: w= 1.98, v= 100

A: w= 2, v= 40

D: w=5, v=95

B: w= π, v= 50

E: w= 3, v= 30

15 nodes checked!

Branch-and-Bound - Ex: 0-1 Knapsack Problem

```
def sort_heaviest_top(ws, vs):
    sorted_items = sorted(list(zip(ws, vs)), key = lambda x: x[0], reverse = True)
    sorted_ws = [w for (w, _) in sorted_items]
    sorted_vs = [v for (_, v) in sorted_items]
    return sorted_ws, sorted_vs

def sort_density_top(ws, vs):
    sorted_items = sorted(list(zip(ws, vs)), key = lambda x: x[1]/x[0], reverse = True)
    sorted_ws = [w for (w, _) in sorted_items]
    sorted_vs = [v for (_, v) in sorted_items]
    return sorted_ws, sorted_vs
```

Branch-and-Bound - Ex: 0-1 Knapsack Problem

```
def bnb_knapsack(ws, vs, W):
    ws, vs = sort_density_top(ws, vs)
    s = []                 # empty stack
    n = len(ws)
    max_profit = 0  # keep tracking the profit of the best solution so far
    s.append((0, 0, W))      # state/node: (level, current profit, current capacity)
    while s != []:
        (l, v, w) = s.pop()
        if w >= 0 and bound(l, v, w, ws, vs) > max_profit:  # if promising
            max_profit = max([max_profit, v])
            if l < n:              # if l == n, we stop since we have inspected last item
                leave_i = (l+1, v, w)
                take_i = (l+1, v + vs[l], w - ws[l])
                s.append(leave_i)       # push "leave item i" child node
                s.append(take_i)        # push "take item i" child node
    return max_profit
```

Running Time?

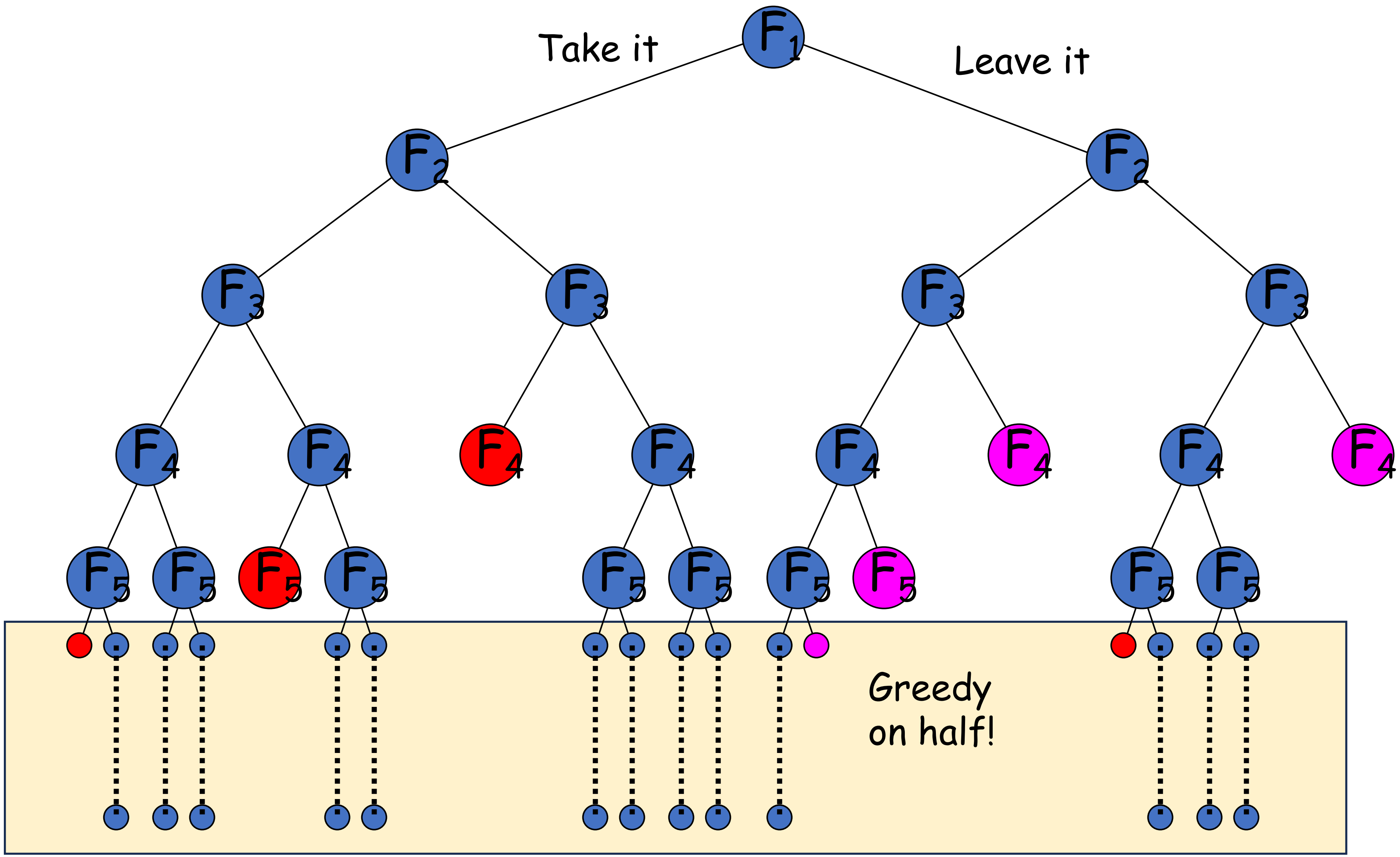
- Backtracking algorithms for problems such as the 0-1 Knapsack problem are still **exponential-time** in the worst case!
- They are useful because they are **efficient for many large instances**.
- Backtracking algorithms depend on the data they are given!
  - With one set of data, we might have to complete a DFS without eliminating any branches.
  - With another set of data, we might eliminate most of the branches

Greedy + Backtracking

- Suppose half the inputs to our knapsack problem all have the same weight.
  - If all inputs had the same weight, we could implement a greedy solution, highest value first.
  - However, since half do not, we cannot use greedy alone to find optimal solution
- Combine backtracking approach with greedy to find optimal solution more quickly.

Example:

A w=10 v=50
B w=20 v=40
C w=5 v=10
D w=5 v=5
E w=15 v=30
F w=15 v=20
G w=15 v=15
H w=15 v=10



Example:

F1 w=.. v=..
F2 w=.. v=..
F3 w=.. v=..
F4 w=.. v=..
F5 w=.. v=..
F6 w=15 v=..
F7 w=15 v=..
F8 w=15 v=..
F9 w=15 v=..
…

Running Time?

- Backtrack on half the inputs
- At leaves, apply greedy strategy on the other half of the inputs
- Comparison
  - Pure brute force: **$O(2^n)$**
  - Combination Greedy + Backtracking: **$O(n2^{n/2})$**

Running Time?

- Assumptions
  - Suppose **n=50**
  - We can test **1,000,000** solutions/second.
- **2^{50}** would take over 35 years
- **2^{25}** can be generated in half an hour
  - Plus marginal time to generate greedy solution for each of the **2^{25}** solutions

DP + Backtracking

- Suppose half the inputs to our knapsack problem have **small integral weights/values** while the other half have **real weights/values**.
- How can we combine approaches to solve this efficiently?
- Dynamic Programming on half!

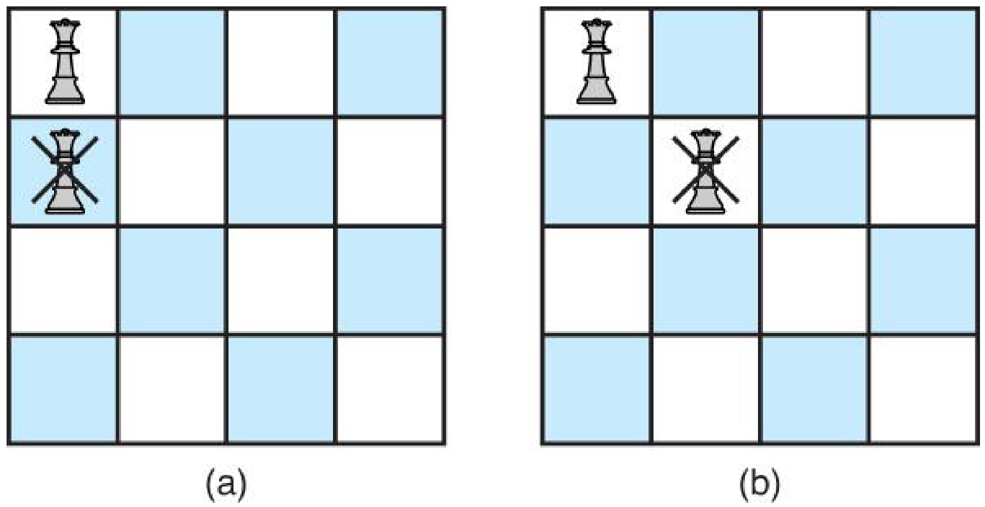
N-Queens Problem

Position n queens on an **n x n** chessboard so that no two queens threaten each other.

- The **criteria** is that no two queens can be in the same row, column, or diagonal;
- The **sequence** for the problem is the **n** positions in which the queens are placed;
- The **set** for each choice is 2^n possible positions on the chessboard.

4-Queen Problems

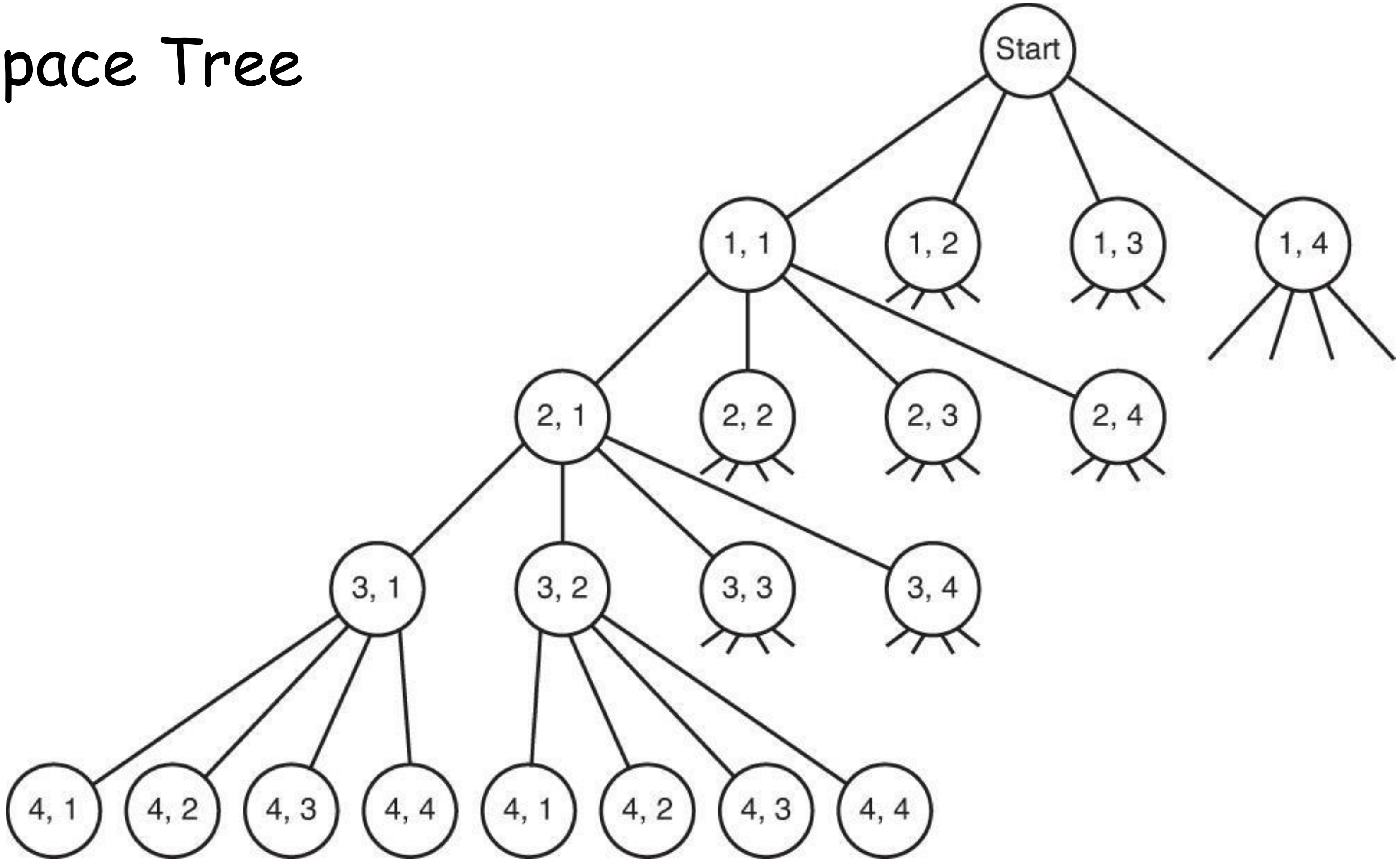
- Pure brute force search
  - 16 squares on a 4 by 4 chessboard
  - Leads to 16 * 15 * 14 * 13 = 43,680 possible solutions



- Improvements
  - **At most one queen per row**: 4^4 = 256 possible solutions
  - Backtracking: If two queens already attack before final queen placed, backtrack

4-Queen Problems

State Space Tree

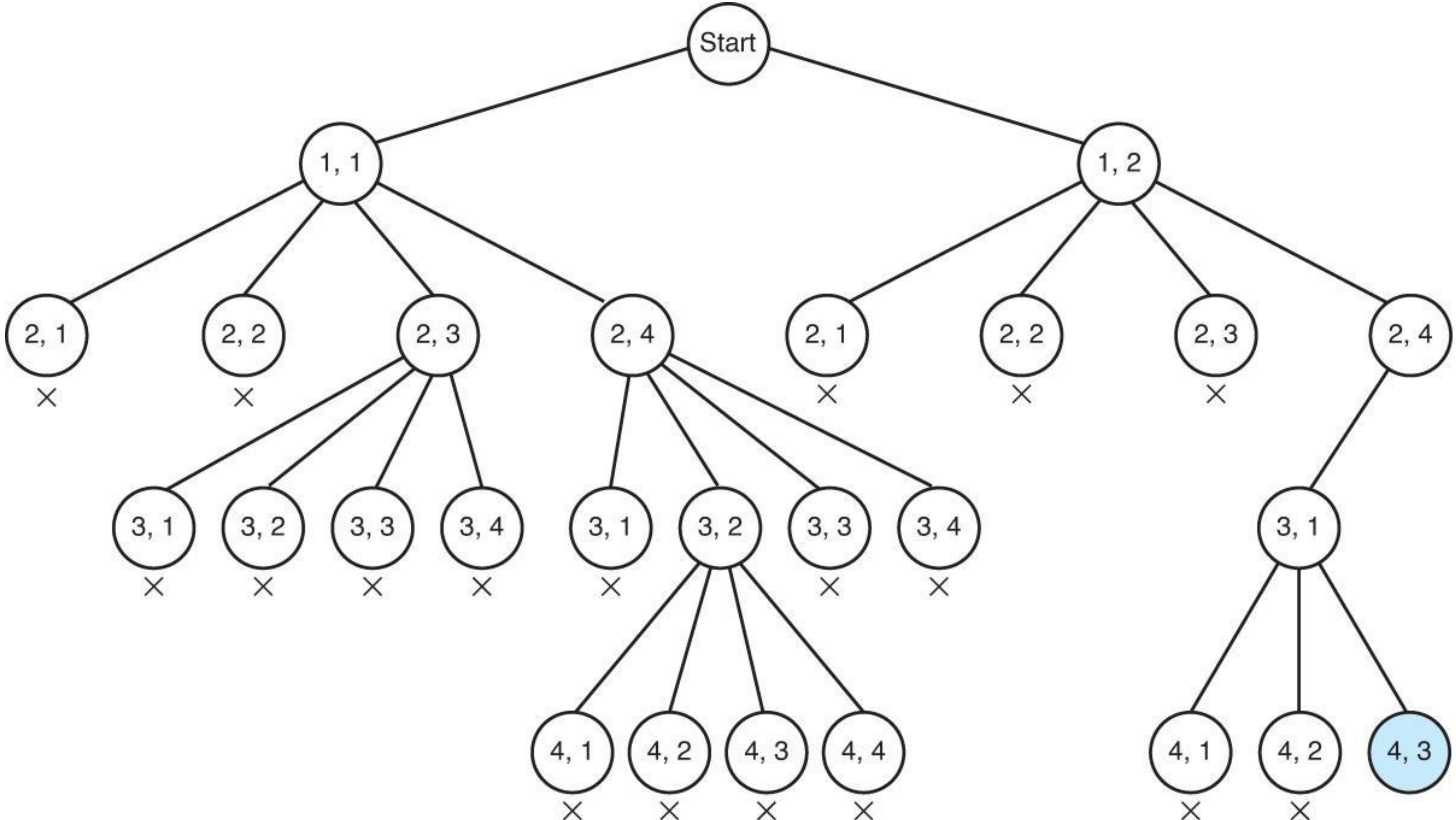


4-Queen Problems

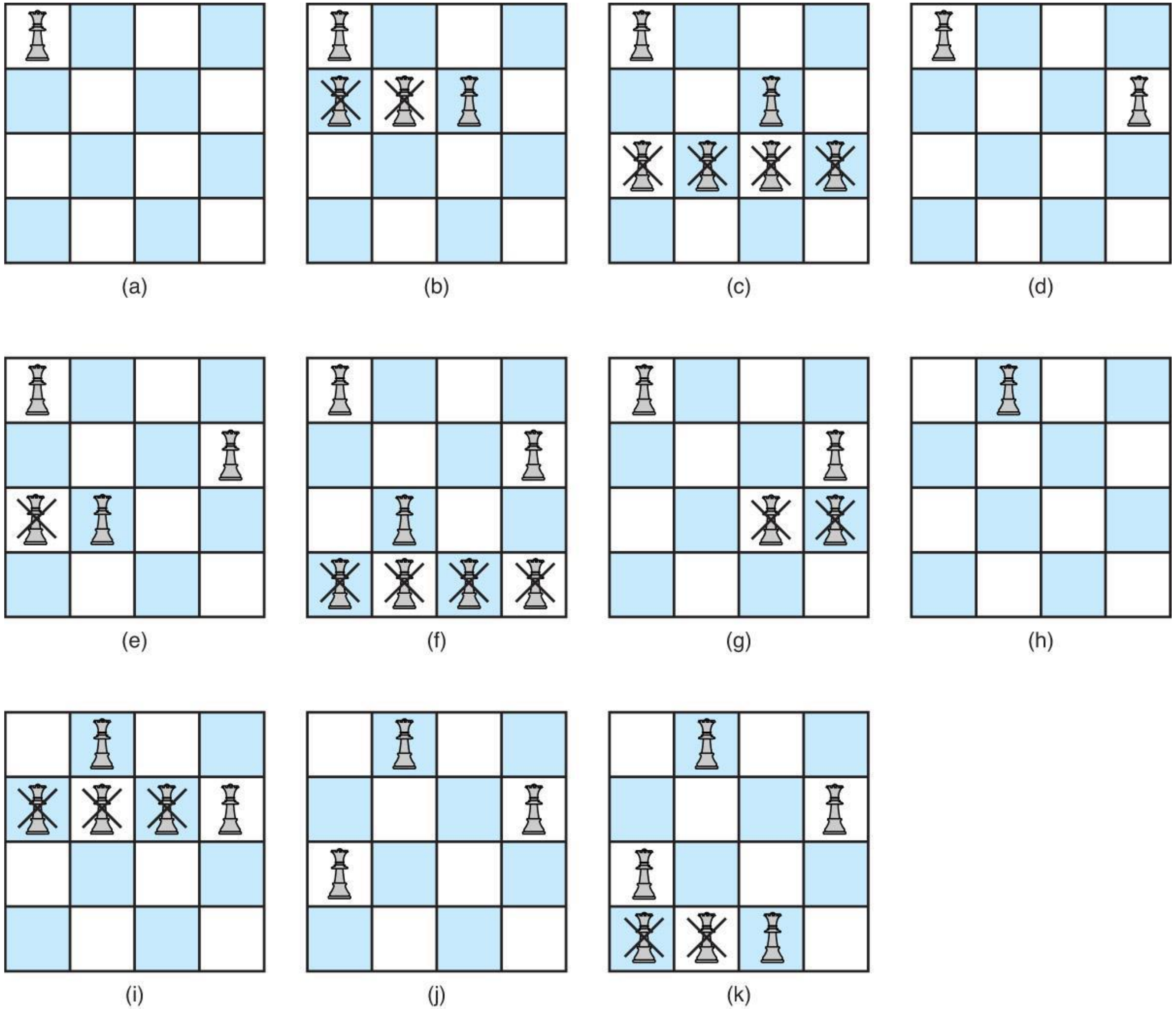
A node is **non-promising** if

- The queen in the ***k*-th** row threatens the queen in the ***i*-th** row along the same column:
  - **column(i) = column(k)**
- The queen in the ***k*-th** row threatens the queen in the ***i*-th** row along one of its diagonals:
  - **column(i) – column(k) = i – k**
  - **column(i) – column(k) = k - i**

4-Queen Problems



4-Queen Problems



N-Queen Problems

```
def n_queens_recc(board, N, row):
    if row == N:
        return True
    for col in range(N):
        if is_safe(board, N, row, col):
            board[row][col] = 1
            if n_queens_recc(board, N, row + 1):        # check for next row
                return True
            board[row][col] = 0
    return False

def n_queens(N):
    board = [[0 for x in range(N)] for y in range(N)]
    if n_queens_recc(board, N, 0):
        print(board)
```

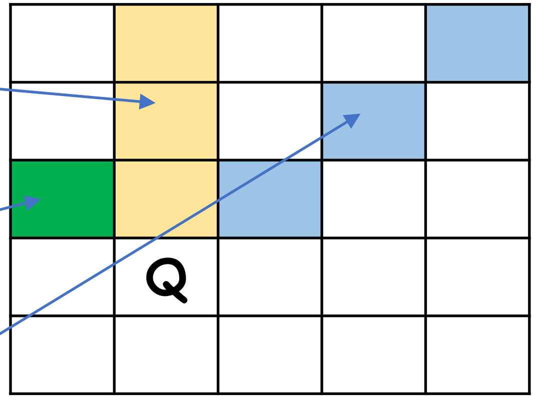
Inefficient implementation of is_safe?

```
def is_safe(board, N, row, col):

    for x in range(row):
        if board[x][col] == 1:
            return False

    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False

    for x, y in zip(range(row, -1, -1), range(col, N, 1)):
        if board[x][y] == 1:
            return False
    return True
```



Running time = O(N)

is_safe in O(1) time?

The idea is to keep **three Boolean arrays that tell us which rows and which diagonals are occupied**.

Each row has already had a unique number. But, **how can we give each diagonal a unique number?** **HINT:**

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

The "\" diagonals
Row – Col + N - 1

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

The "/" diagonals
Row + Col

Exercise!

In the U.S. navy, the SEALS are each specially trained in a wide variety of skills so that small teams can handle a multitude of missions.

If there are *k* different skills needed for a mission, and *n* SEAL members that can be assigned to the team, find the smallest team that will cover all of the required skills.

Andersen knows **hand-to-hand**, **first aid**, and **camouflage**
Butler knows **hand-to-hand** and **snares**
Cunningham knows **hand-to-hand**
Douglas knows **hand-to-hand**, **sniping**, **diplomacy**, and **snares**
Eckers knows **first-aid**, **sniping**, and **diplomacy**

Exercise!

Given a graph G, check whether there is a cycle that visits every vertex of G exactly once and returns to the starting vertex.

Describe a state space tree for this problem; what are promising nodes? How to determine a solution from the tree?