算法设计与分析

Lecture 12: Backtracking

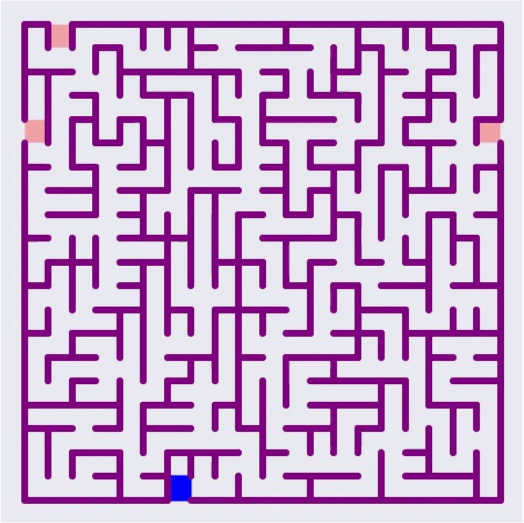
卢杨

厦门大学信息学院计算机科学系

luyang@xmu.edu.cn

Backtracking

- A simple and straightforward strategy to escape from a maze is:
  - Go as deep as possible until reach a dead end.
  - Go back to the last fork and choose another path.
- If we have a sign at the fork to show dead ends, we can avoid that path to save time.
- This is the idea of backtracking (回溯). It is a refinement of the brute force approach by avoiding dead ends in advance.

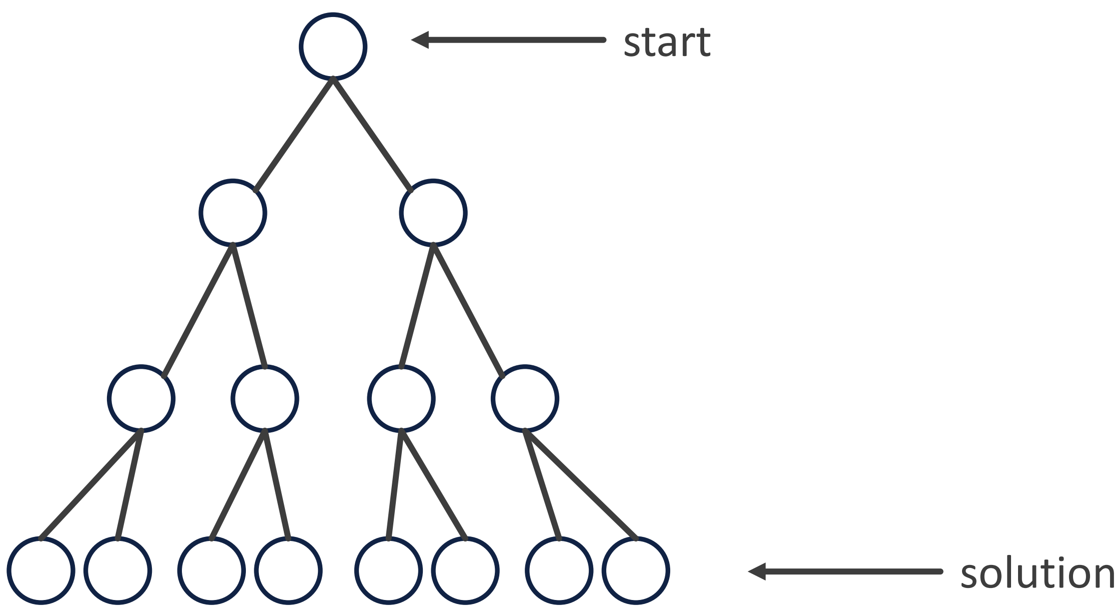


A maze

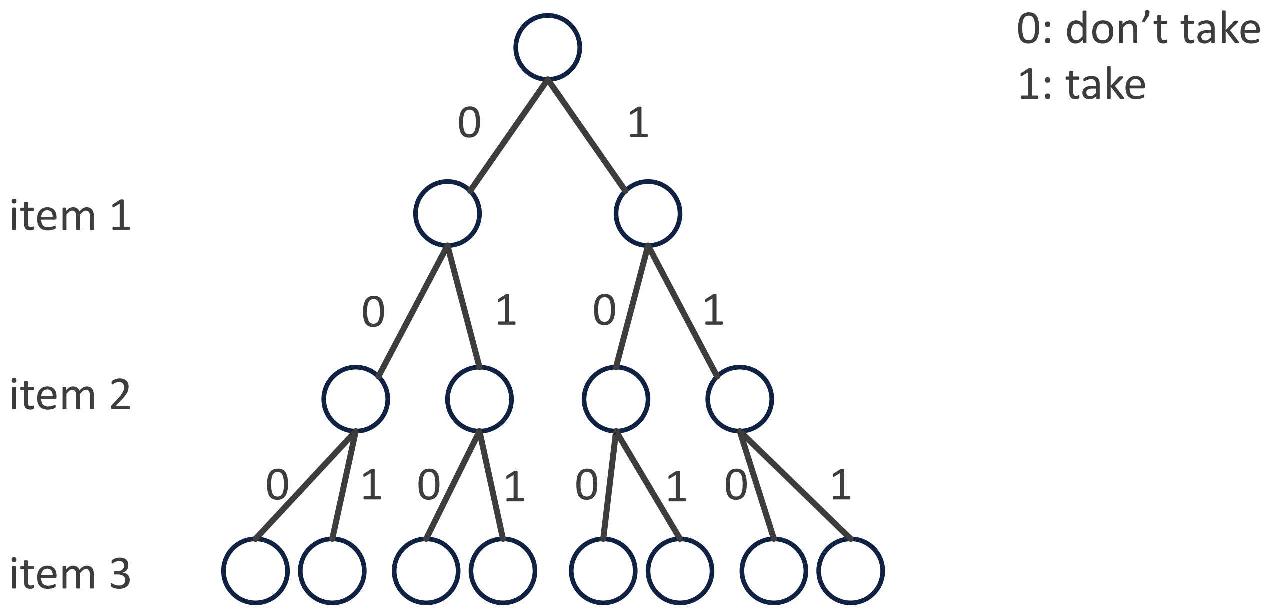
Image source: https://gfycat.com/ru/babyishtightjanenschia-mazesolving-backtracking

Backtracking

- Given the an optimization problem, we usually make a sequence of decisions. It can be represented as a tree.
- We start from the root and the solutions are the leaves.



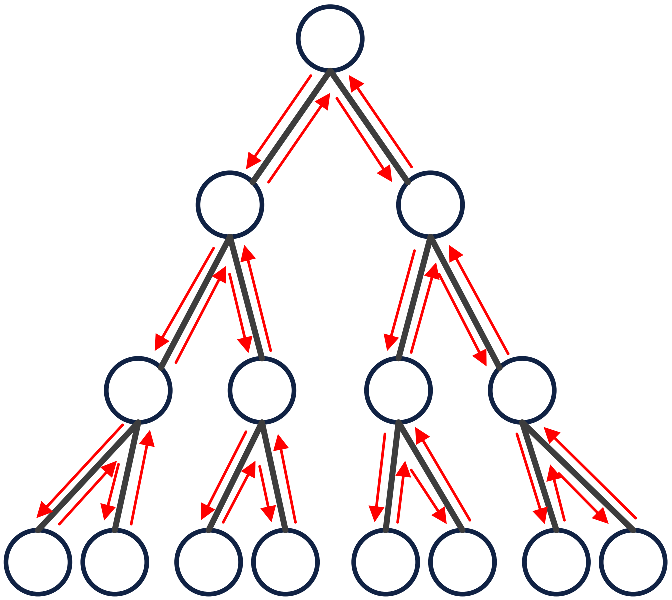
Backtracking

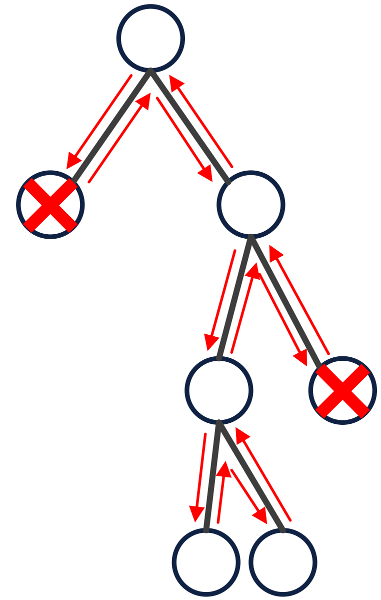


Solution space for 0/1 knapsack problem with 3 items

DFS vs. Backtracking

If we know that going along this branch has no hope, we don't need to try! It will save a lot of time.





Backtracking

- Backtracking is all about HOPE!
- We only continue to search solutions only if there is still hope!

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系
Computer Science Department of Xiamen University

Image source: https://giphy.com/explore/there-is-still-hope

Backtracking

- In the backtracking method, the solutions are represented by vectors $(x_1, x_2, \dots, x_n)$.
- In step $i + 1$, we start from a partial solution $(x_1, x_2, \dots, x_i)$ and try to extend it by adding another element x_{i+1}.
- After extending it, we will test whether $(x_1, x_2, \dots, x_i, x_{i+1})$ is still possible as a partial solution (check hope).

Backtracking

The steps involved in the backtracking method are:

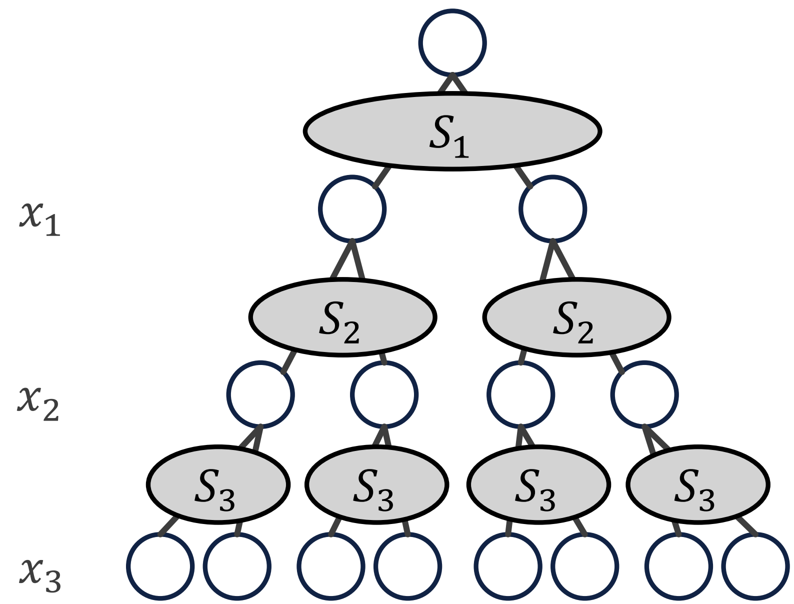
1. Define a solution space (解空间) for the problem. This space must include at least one (optimal) solution to the problem.
   - If S_i is the domain of x_i, then $S_1 \times S_2 \times \cdots \times S_n$ is the solution space of the problem.
   - Generally, the solution space is very huge, so the cost of searching a objective solution are often unimaginable.
   - For backtracking to be efficient, we must prune (剪枝) the search space.

Backtracking

2. Organize the solution space so that it can be searched easily. The typical organization is either a graph or a tree.
3. Searched the solution space in a DFS manner and avoid moving into subspaces that cannot possibly lead to the answer.

Solution Space Tree

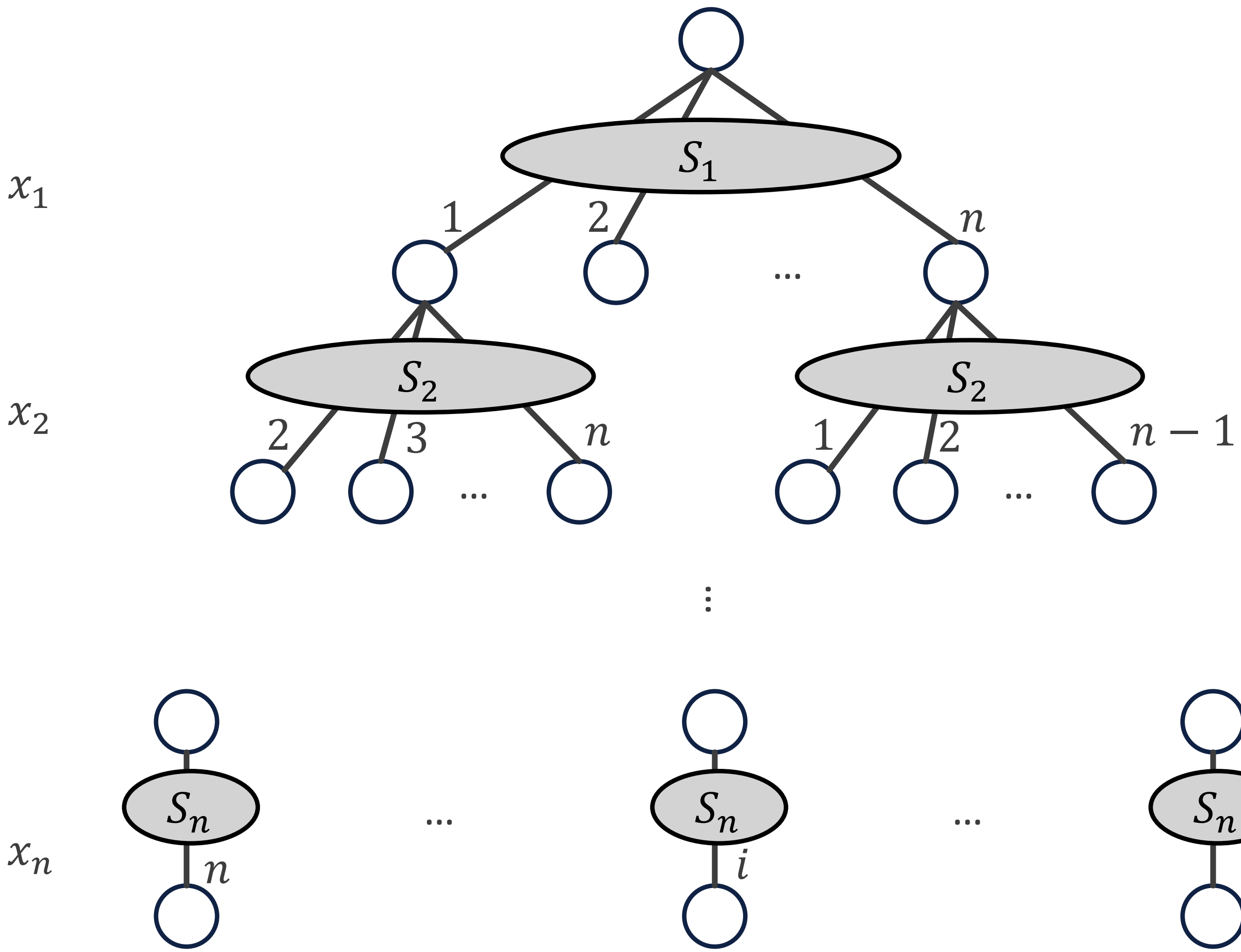
- We set up a tree structure such that the leaves represent members of the solution space.
- So we organize solution space as a solution space tree (解空间树).
- Backtracking can easily be used to iterate through all subsets or permutations of a set.



Example

- When the problem asks for an n-element permutation that optimizes some function, the solution space tree is a permutation tree.
- How many permutations are there of an n-element set?
  - There are n choices for x_1.
  - There are $n - 1$ choices for x_2.
  - …
  - There is only 1 choices for x_n.

S_{i+1} depends on the choice of x_i



General Backtracking Template

```
Backtrack(i)
    if i > n then Update(x)
    else
        for each a ∈ S_i do
            x_i ← a
            if C(i) and B(i) then
                Backtrack(i + 1)
```

Reaching leave means that it is a feasible solution.

Hope checking condition, key of backtracking. Without it, it is just brute-force.

Pruning

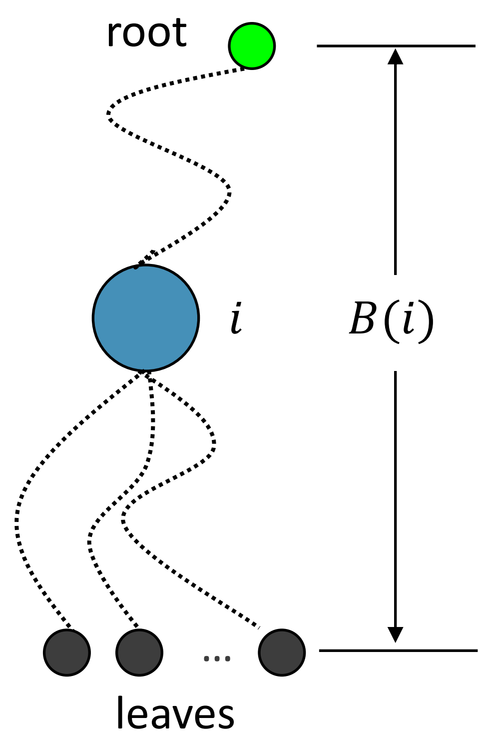
- In backtracking, we have a constraint function (约束函数) $C(i)$ and a bounding function (限界函数) $B(i)$, to prune invalid branches and to focus the search on branches that appear most promising.
  - Keep in mind, we don't waste time on hopeless branches.
- In order to improve the performance of search, applying backtracking requires specifying at least the following three points:
  - How to choose an the constraint function.
  - How to compute upper bounds (for maximum problem) and lower bounds (for minimum problem).
  - How to make use of the constraint function and the bounding function to prune.

Constraint Function

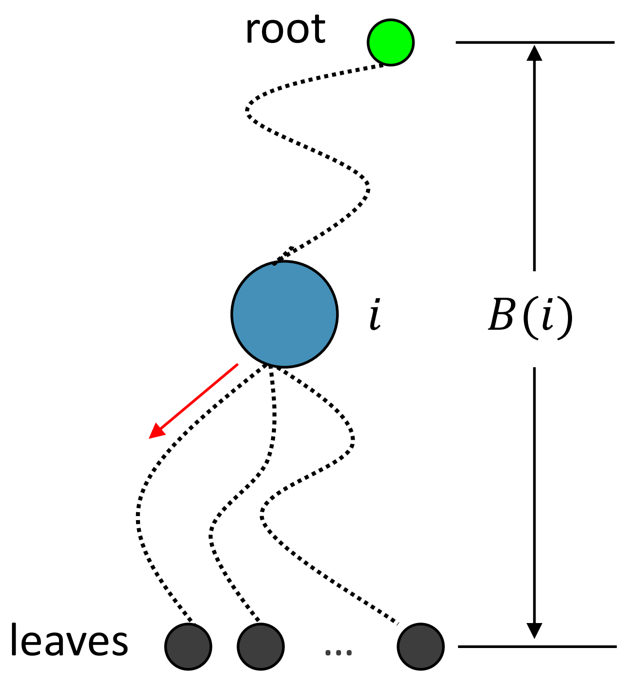
- Constraint function is to check the feasibility of the current solution.
- Usually, it can be easily built by the problem requirement. For example:
  - 0/1 knapsack problem: check if adding the next item exceeds W.
  - Permutation problem: check if the number has been selected.
  - Hamiltonian cycle problem: check (1) if next vertex is connected to the current vertex; (2) if the last vertex is connected to the first vertex; (3) if there exist duplicated vertex in the path.
  - Coloring problem: check if the color for the next vertex is same as its neighbors.

Bounding Function

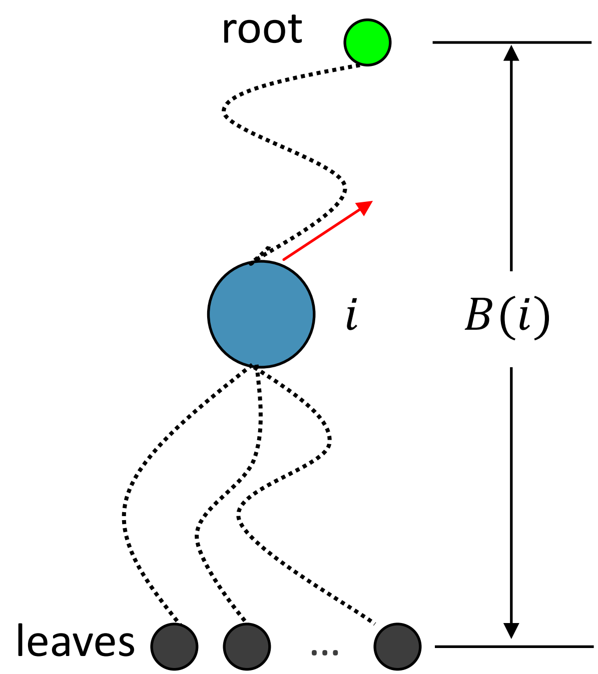
- Bounding function is for optimization problem.
- For maximization problem, it calculates the upper bound of this branch $B(i)$ and compare with the existing best solution $bestc$.
  - If $B(i) > bestc$, there is still hope, keep searching!
  - If $B(i) \leq bestc$, all solutions along this branch will not better than the existing best solution, stop!



Bounding Function



$B(i) > bestc$, go ahead, there is still hope!



$B(i) \leq bestc$, go back, it is hopeless!

CONTAINER LOADING PROBLEM

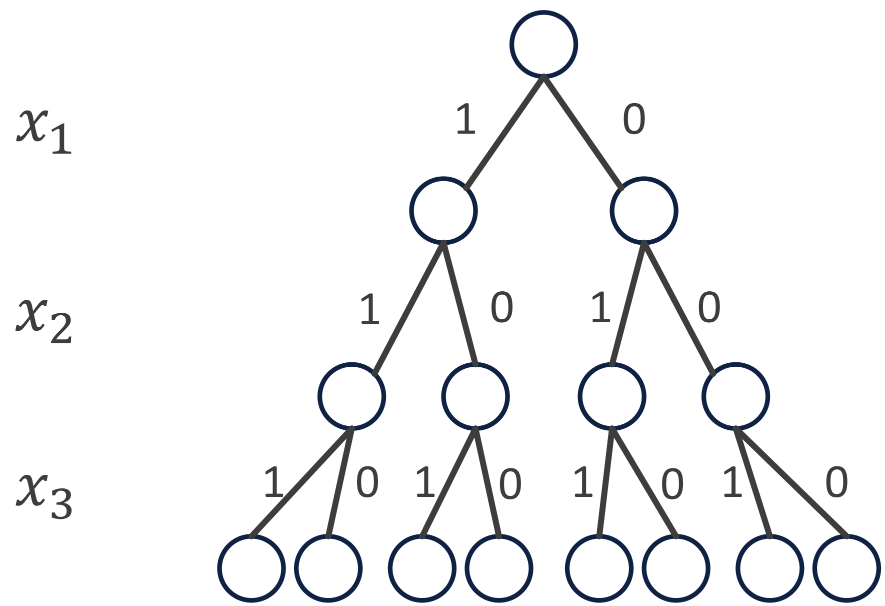
Container Loading Problem

- Given n containers (集装箱), container i has weight w_i. The ship can hold containers of total weight up to W.
- Container Loading problem is to load as many containers as is possible without sinking the ship.
- Assuming that the solutions are represented by vectors $(x_1, x_2, \ldots, x_n)$, where $x_i \in \{0,1\}$. 1 denotes taking container i and 0 denotes not taking container i.
- The container loading problem can be formally stated as follows:

$$\max \sum_{i=1}^{n} w_i x_i \qquad s.t. \sum_{i=1}^{n} w_i x_i \le W$$

Container Loading Problem

- Each x_i has two options to choose: take and not take.
- Therefore, $|S_i| = 2$ and the size of the solution space is 2^n. It also means that the solution space tree has 2^n leaves.



Solution space tree with $n = 3$

Container Loading Problem

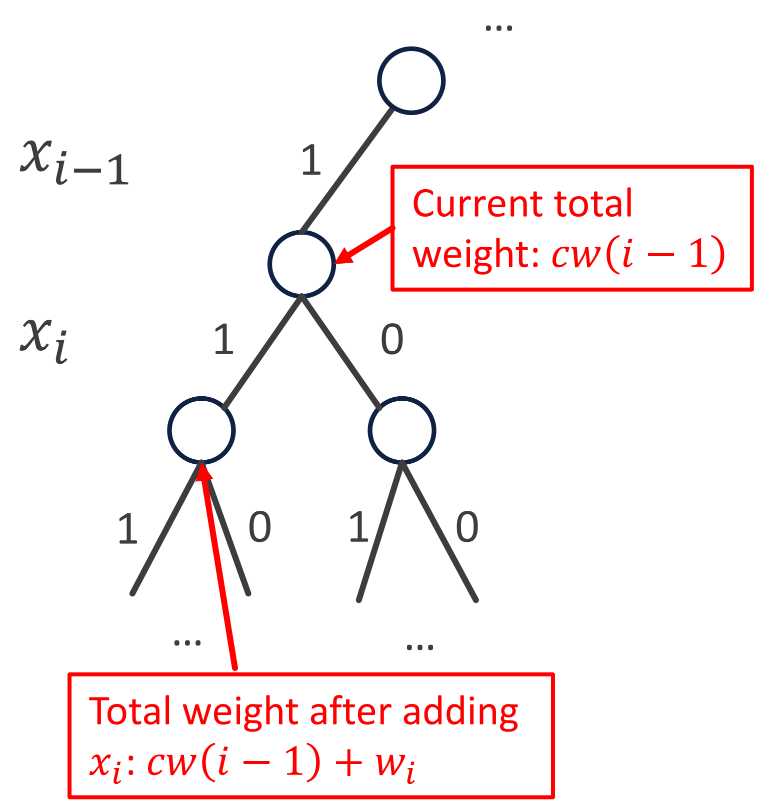
- We first design the constraint function.
- Let $cw(i)$ denote the current weight up to level i, namely

$$cw(i) = \sum_{j=1}^{i} w_j x_j$$

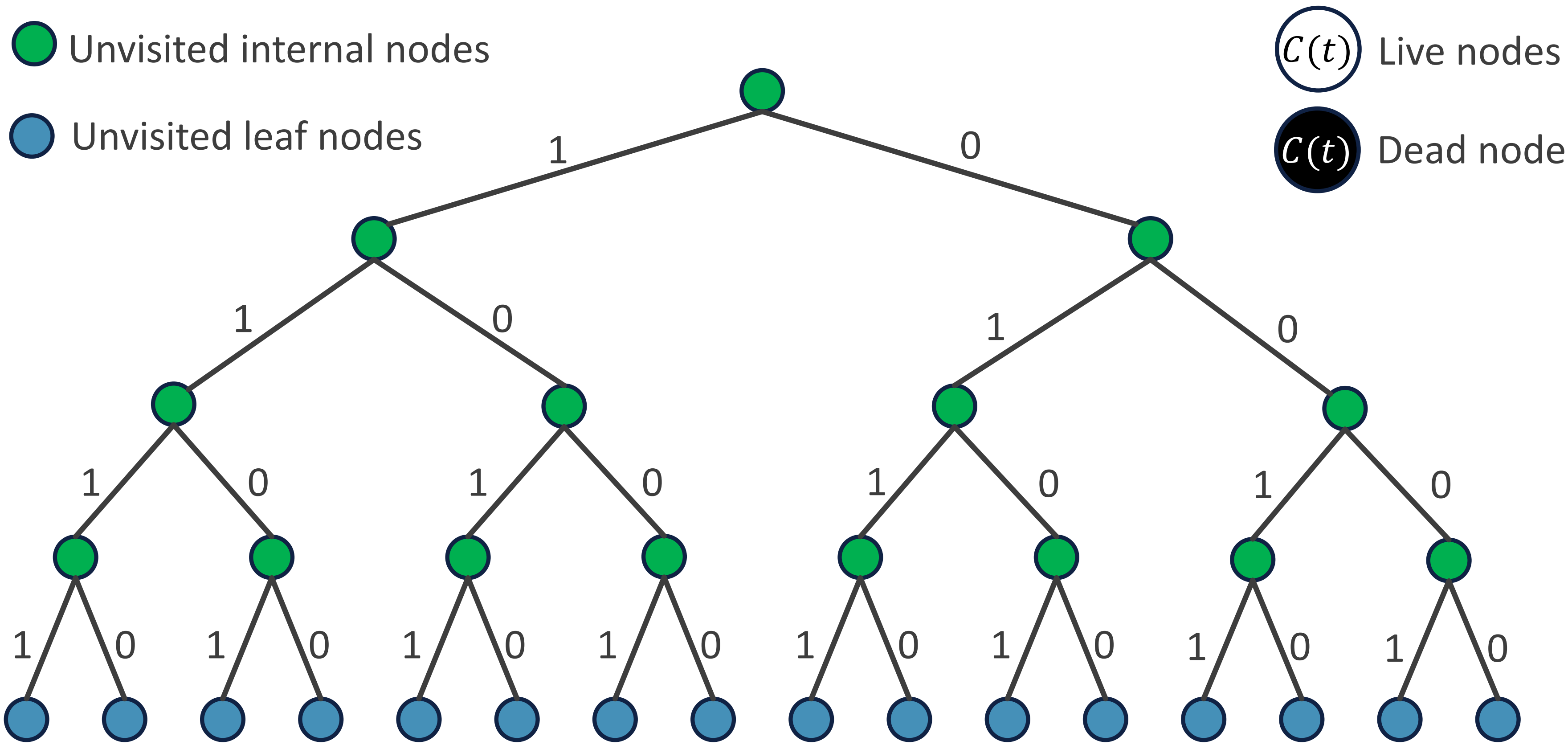
then the constraint function is

$$C(i) = cw(i-1) + w_i$$

- The pruning condition is $C(i) > W$, which means there is no capacity to take container i.

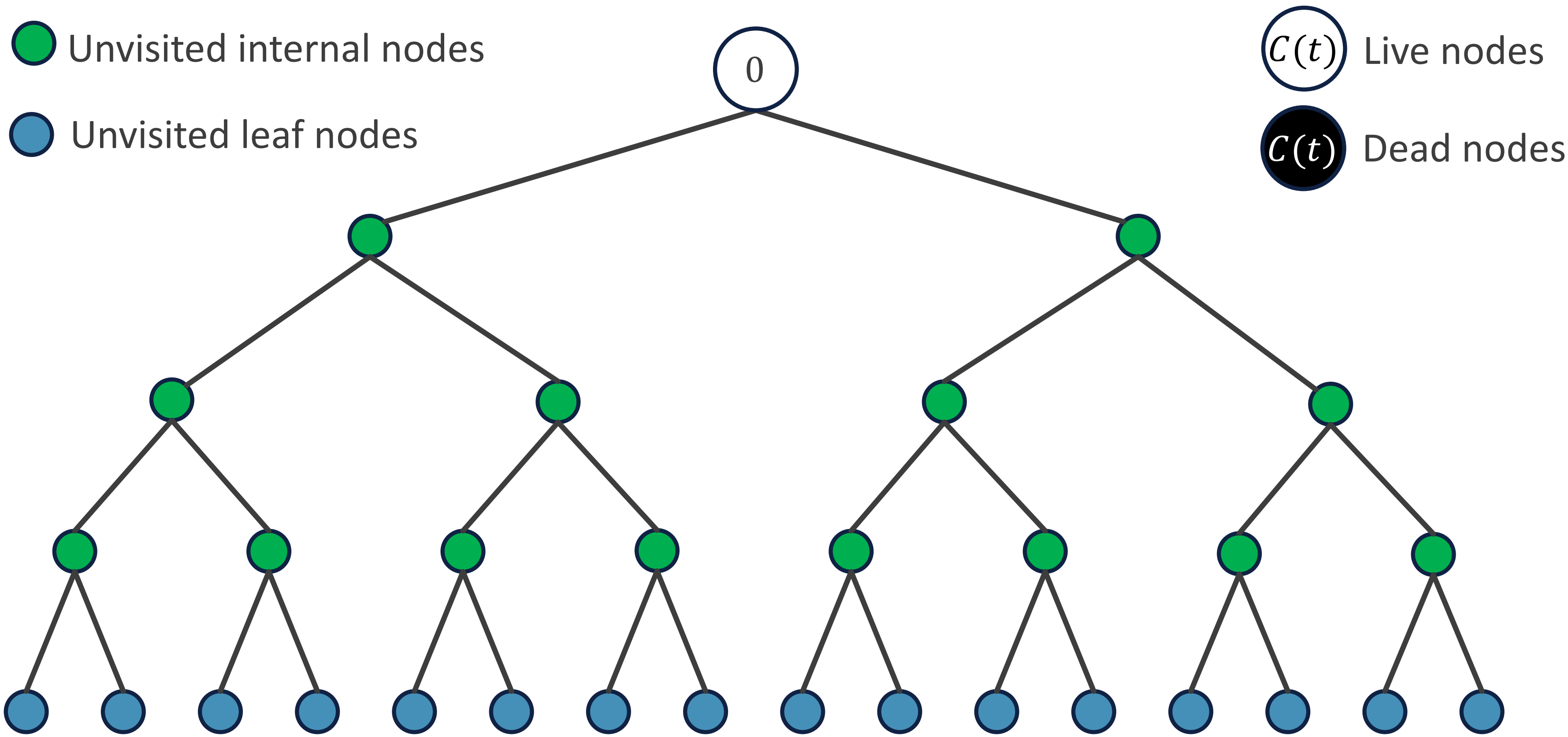


Example



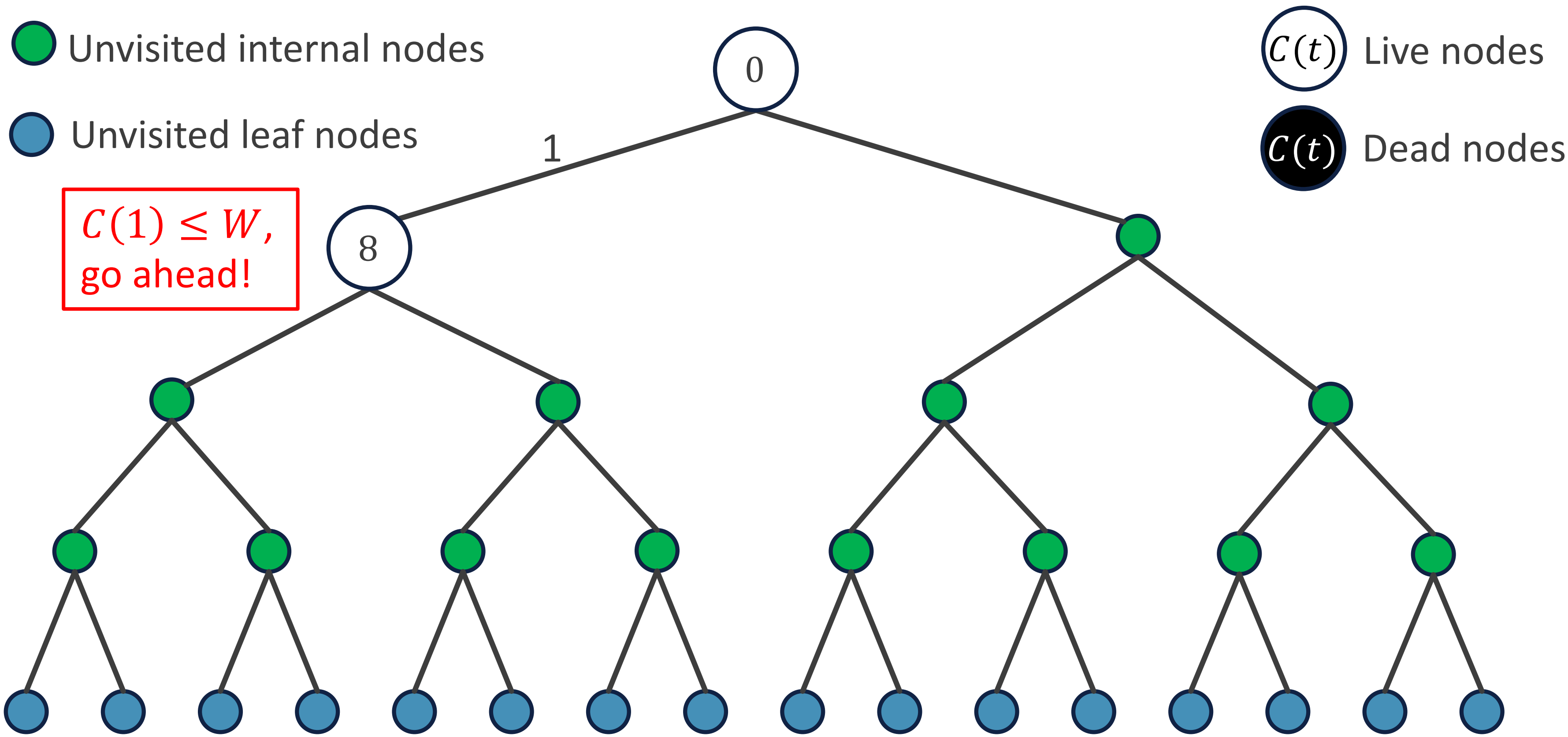
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



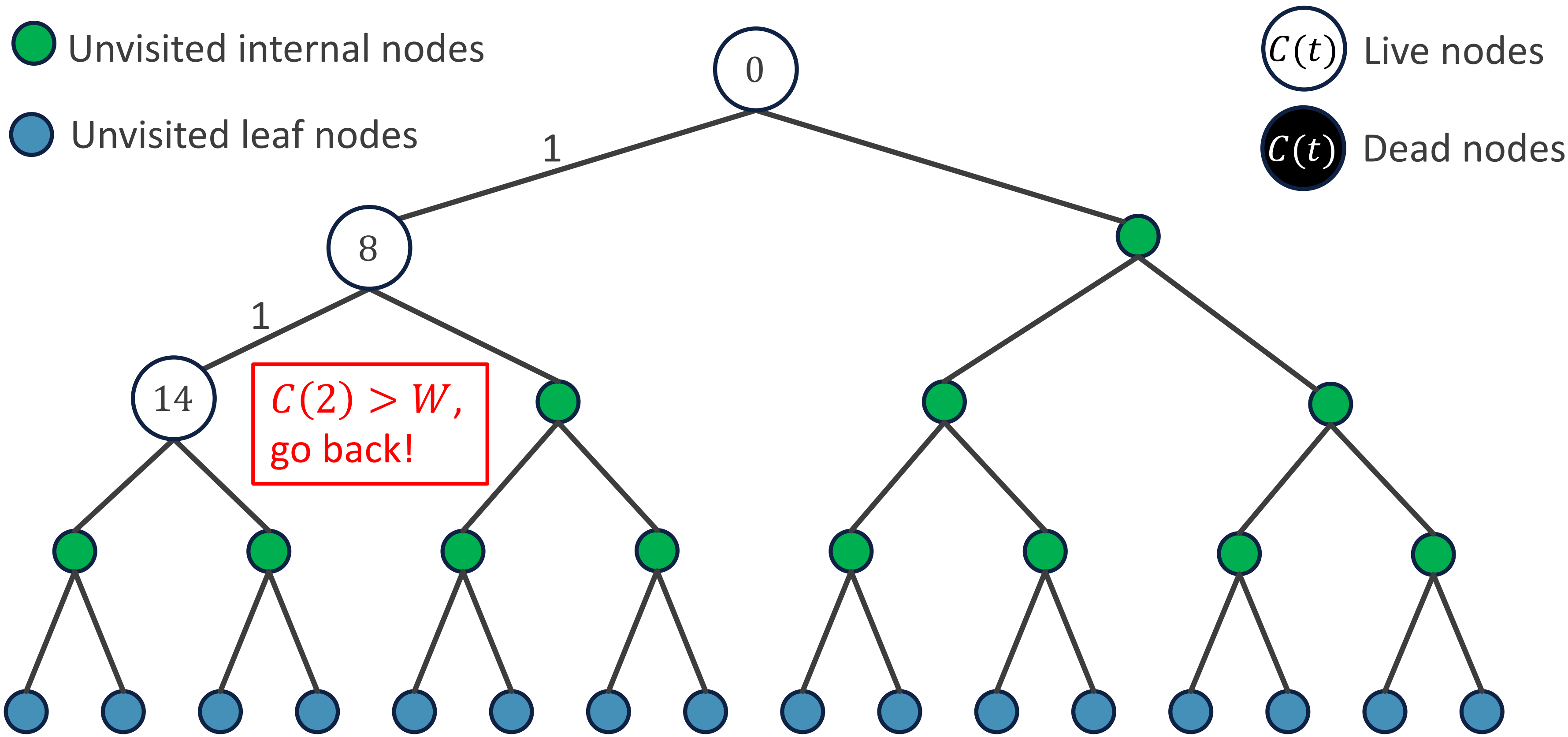
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



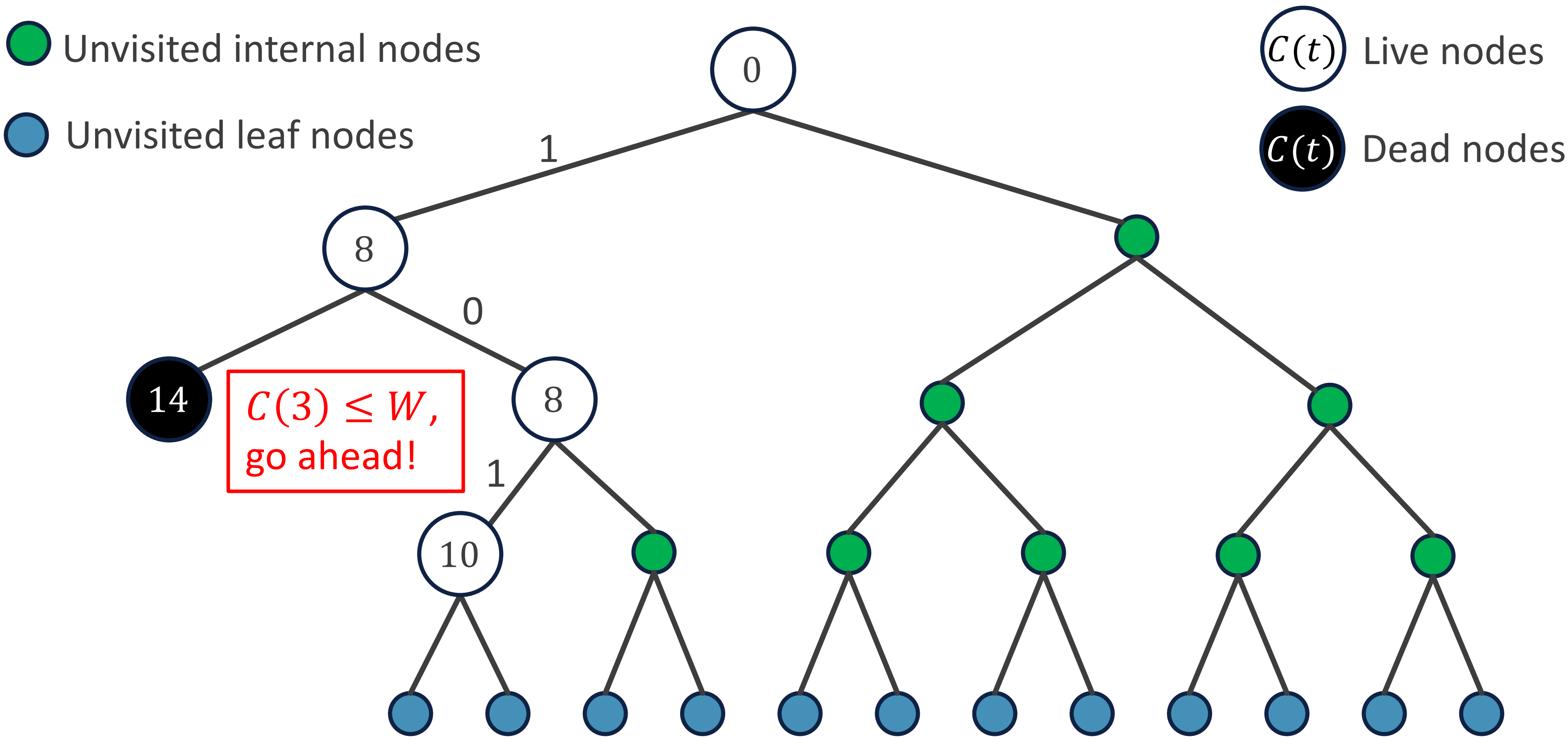
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



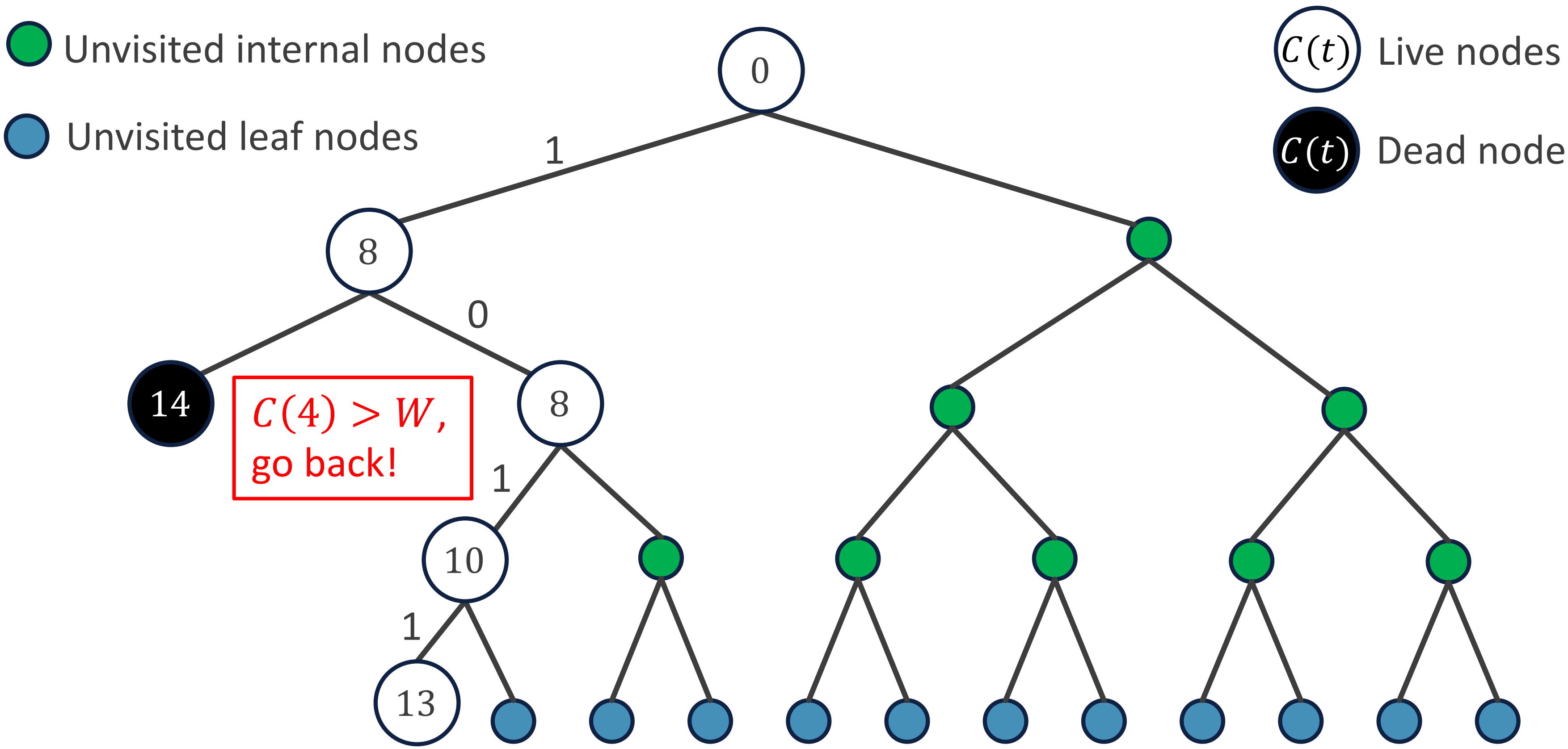
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



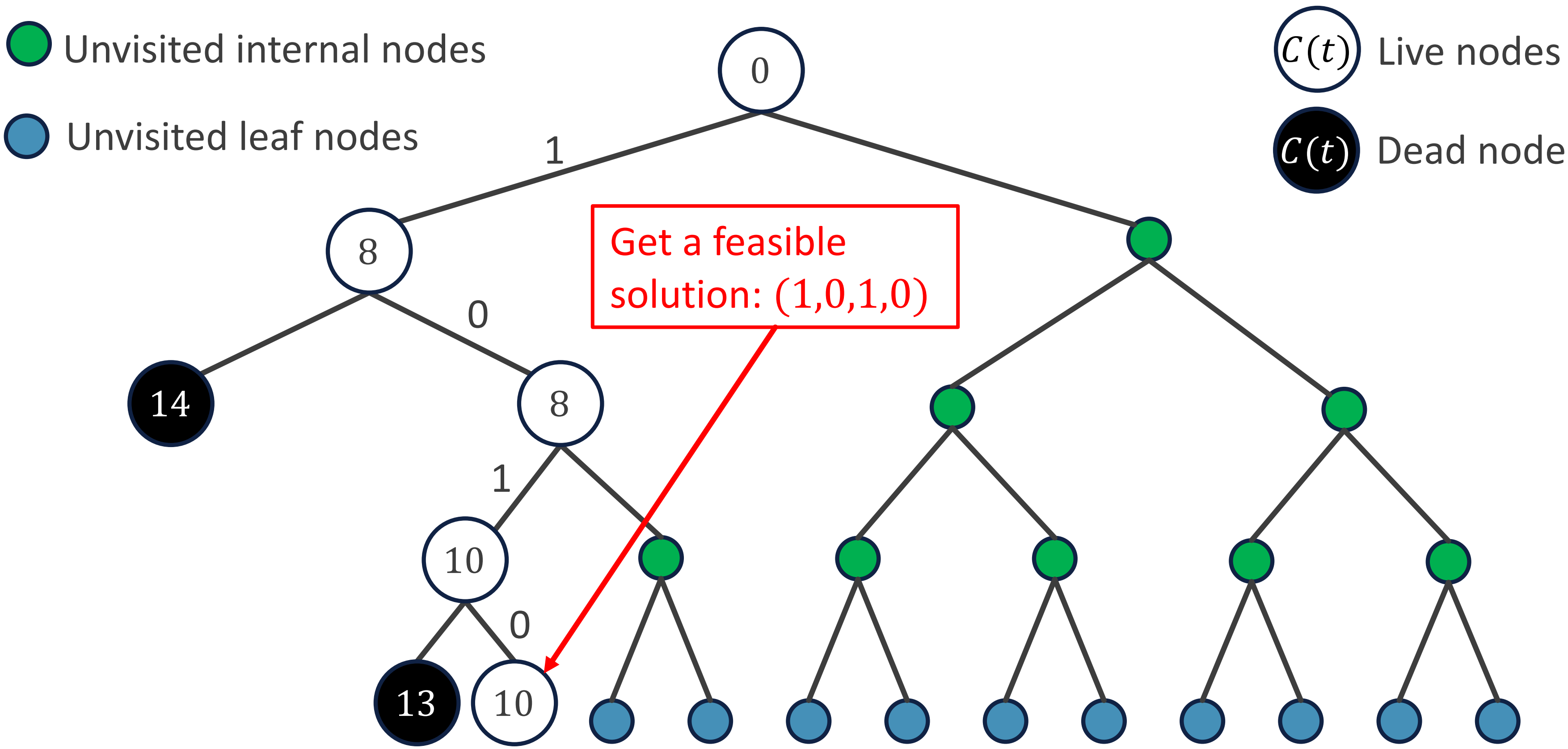
Backtracking for $n = 4, w = [8,6,2,3], W = 12$

Example



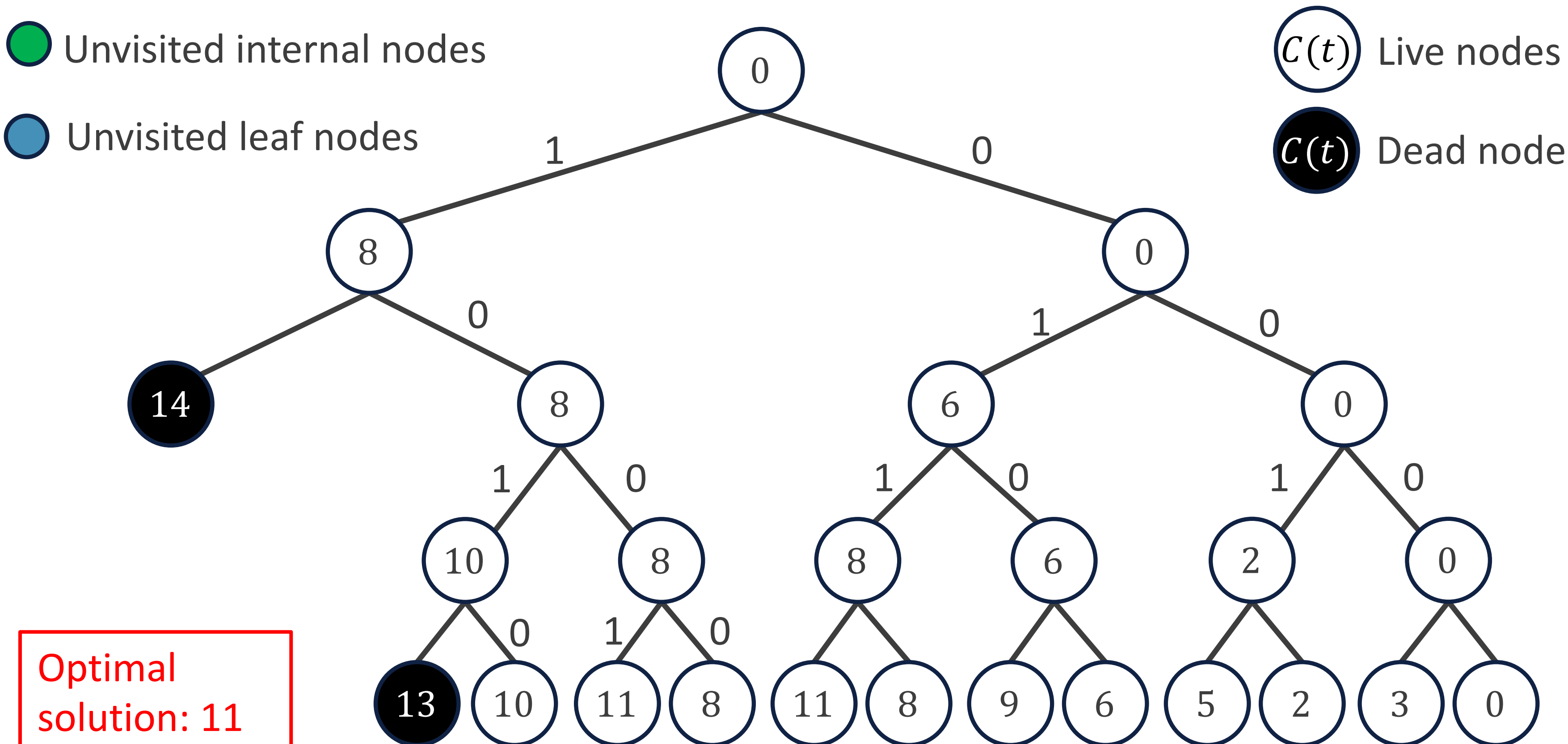
Backtracking for $n = 4, w = [8,6,2,3], W = 12$

Example



Backtracking for $n = 4, w = [8,6,2,3], W = 12$

Example



Optimal solution: 11

Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Pseudocode

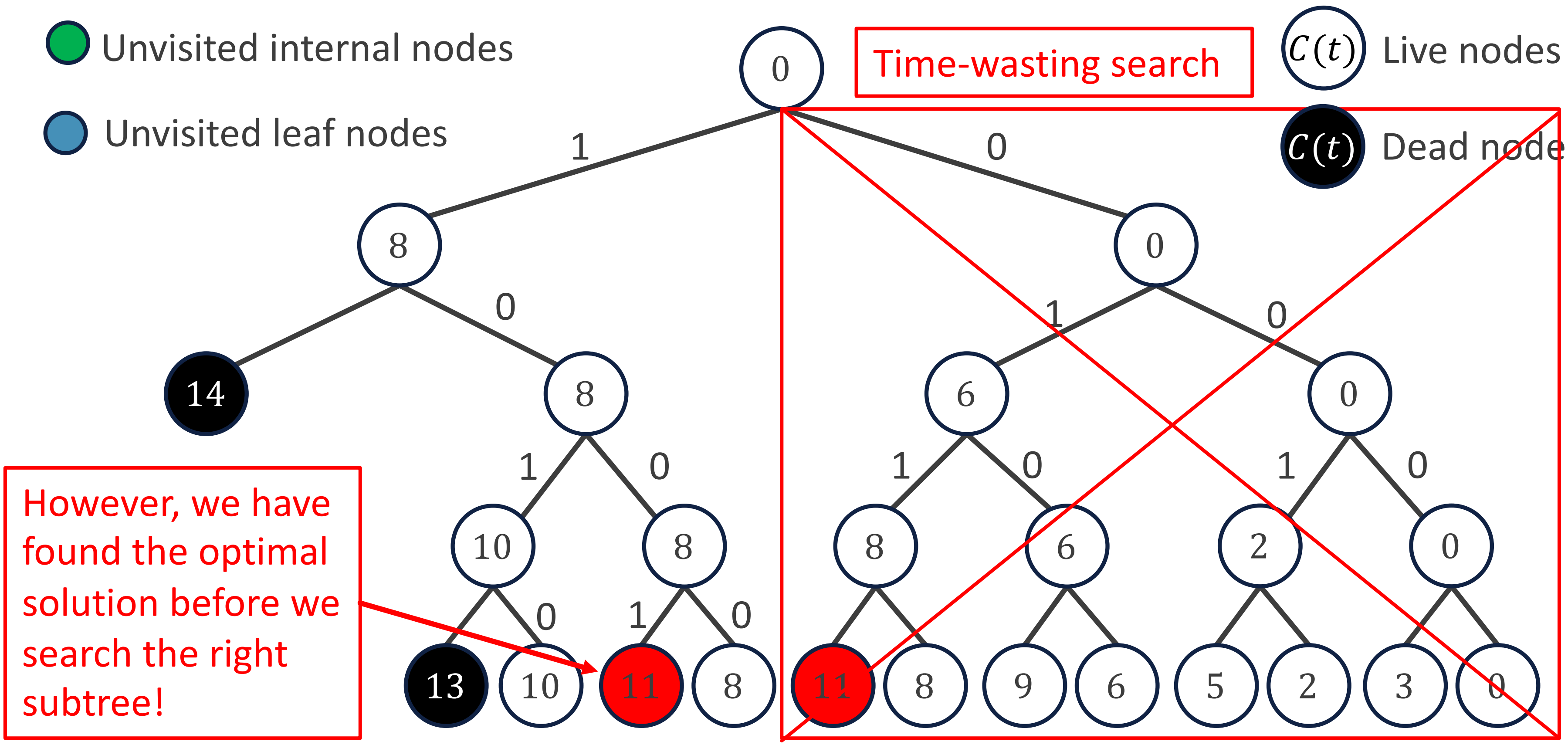
```
BacktrackLoading(i)
  if i > n then
      if cw > bestw then
          bestw ← cw          ← Store best solution so far.
  else
      if C(i) ≤ W then
1         cw ← cw + w[i]
          BacktrackLoading(i + 1)
          cw ← cw − w[i]
0     BacktrackLoading(i + 1)
```

Note: we don't actually build a tree structure. Instead, we simply use recursion.

Go ahead by taking container i.

Subtract the weight of container i before we go back.

Example



Backtracking for $n = 4, w = [8,6,2,3], W = 12$

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系 Computer Science Department of Xiamen University

Example

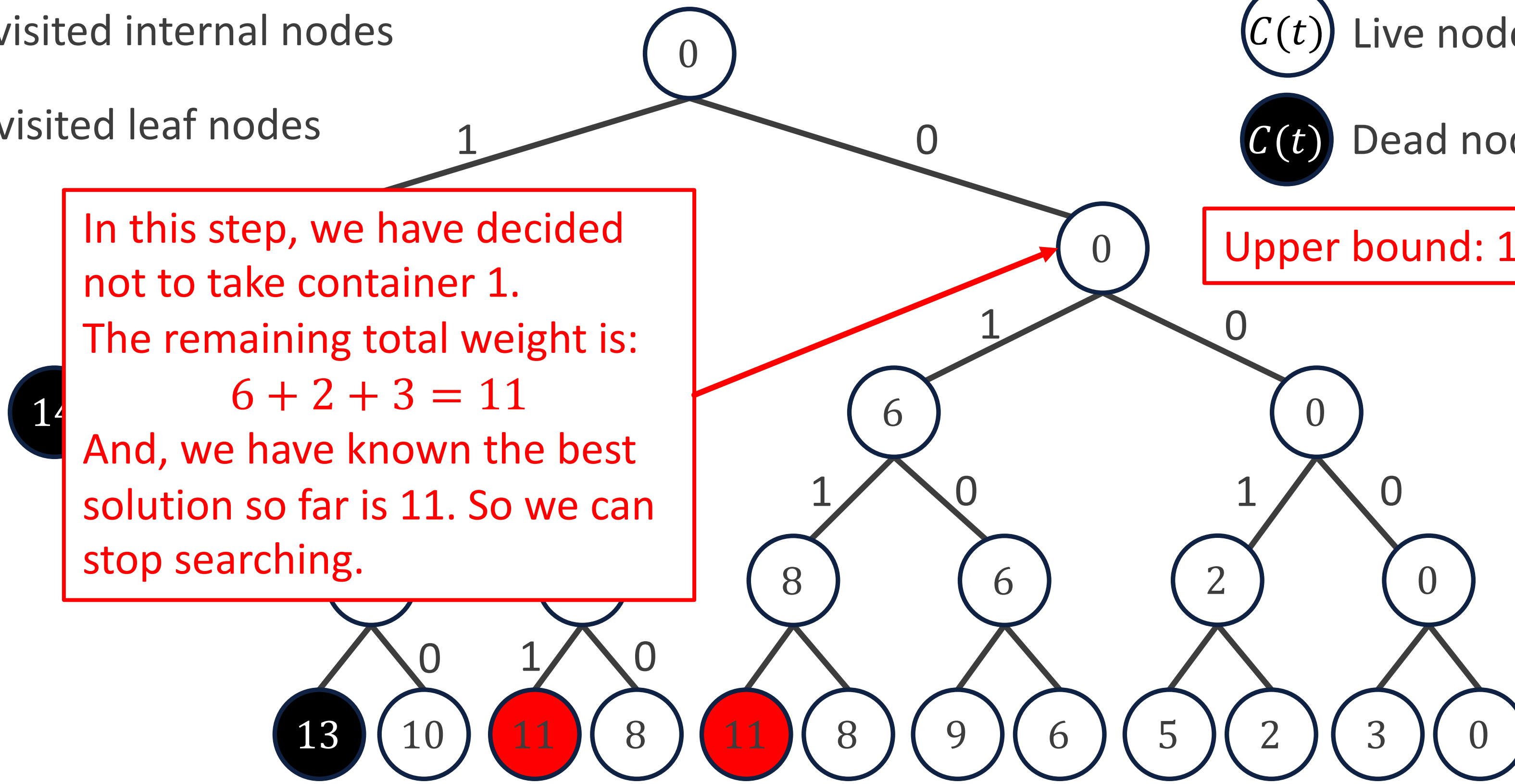
Unvisited internal nodes

Unvisited leaf nodes

$C(t)$ Live nodes

$C(t)$ Dead nodes

Upper bound: 11



In this step, we have decided not to take container 1.
The remaining total weight is:
$$6 + 2 + 3 = 11$$
And, we have known the best solution so far is 11. So we can stop searching.

Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Container Loading Problem

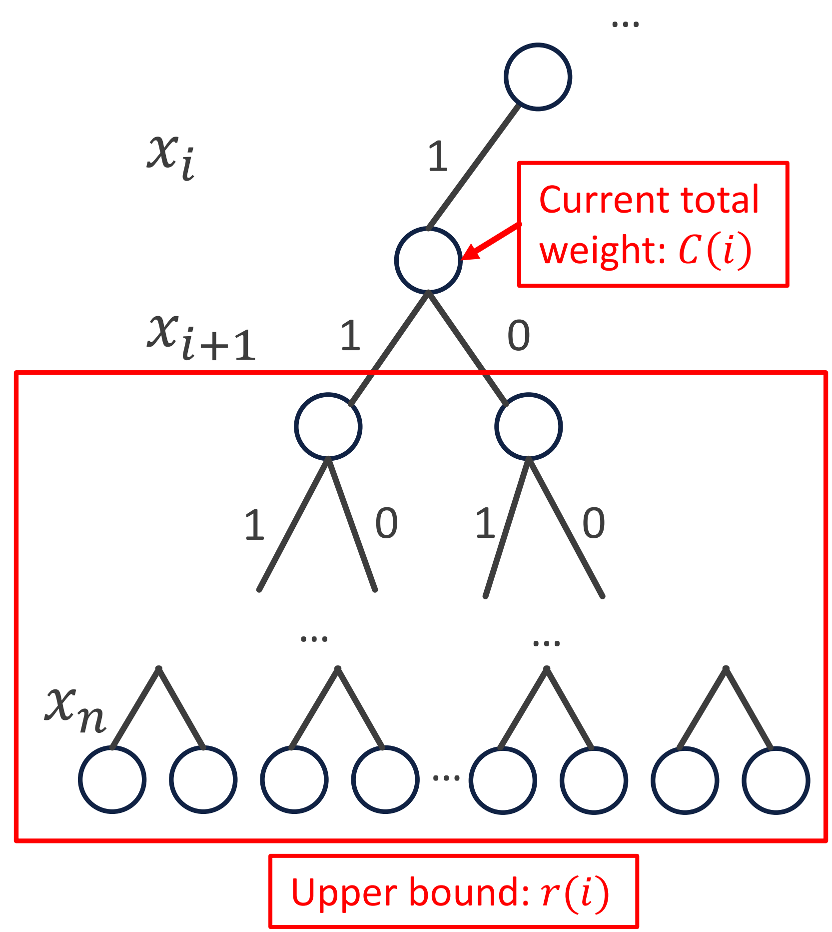
- Now, as an improvement, we add the bounding function:

  $$B(i) = C(i) + r(i)$$

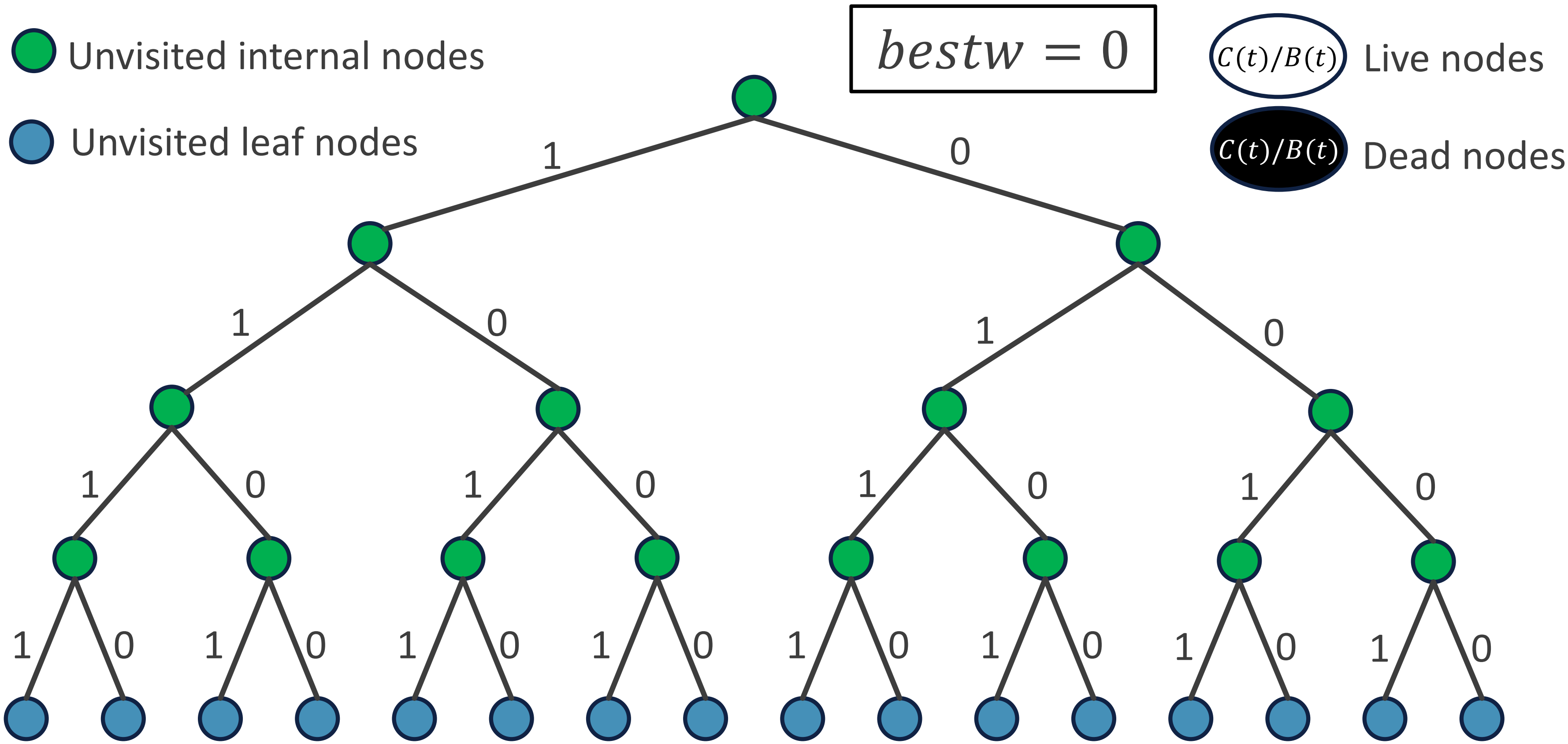
  where, $r(i)$ denotes the weight sum of the remaining containers, namely,

  $$r(i) = \sum_{j=i+1}^{n} w_j$$

- The pruning condition is $B(i) \leq bestw$, which means the continuing searching along this branch will not give better solution.

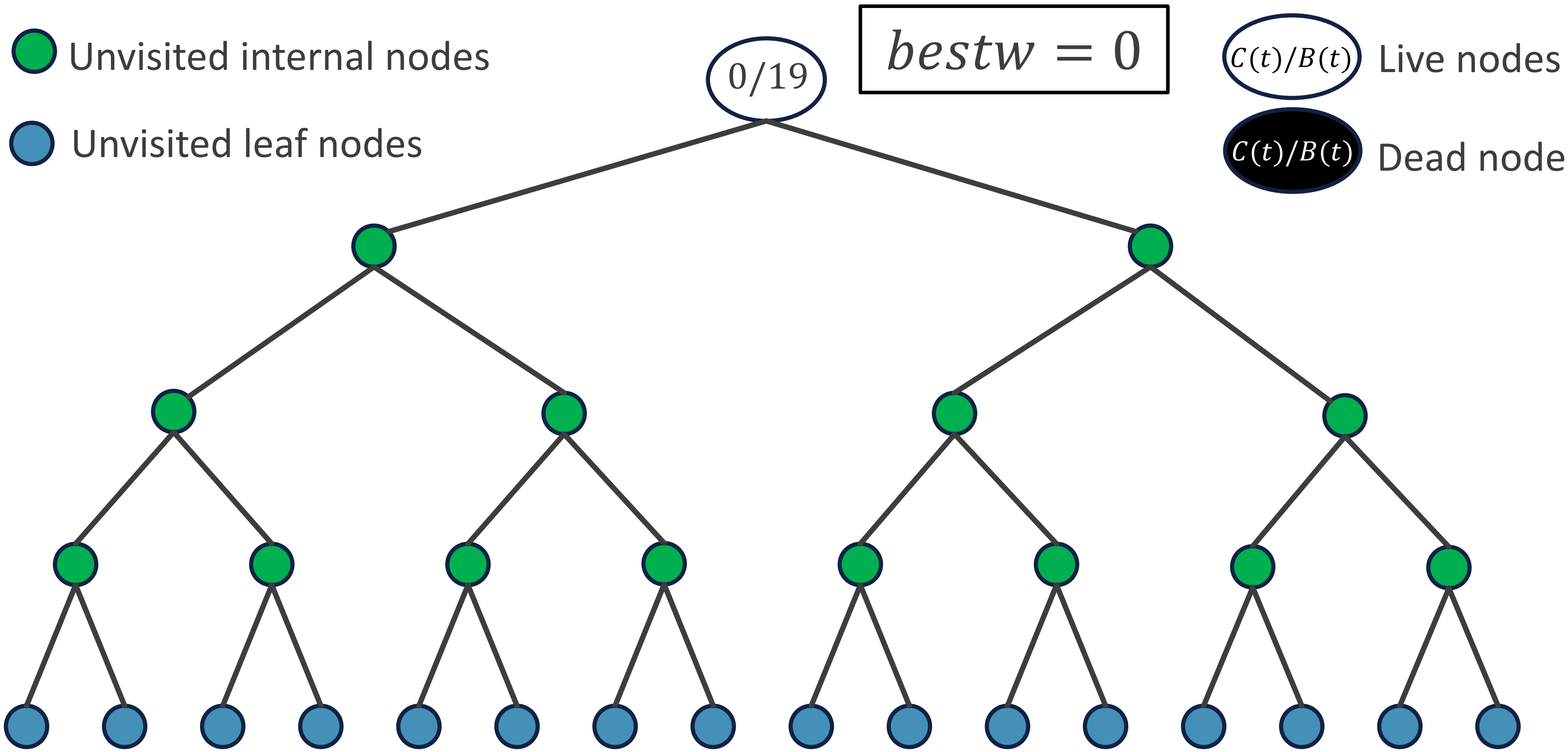


Example



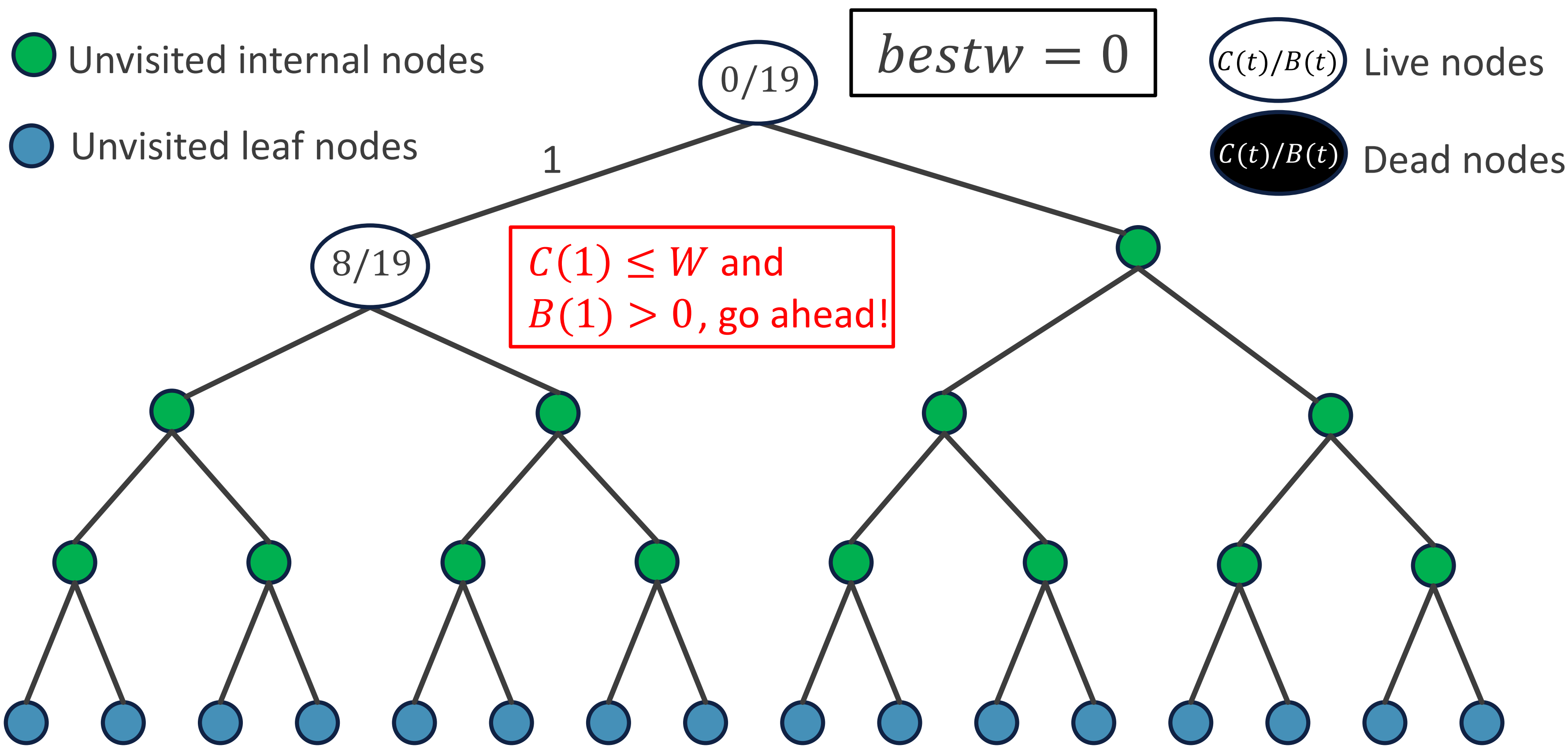
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



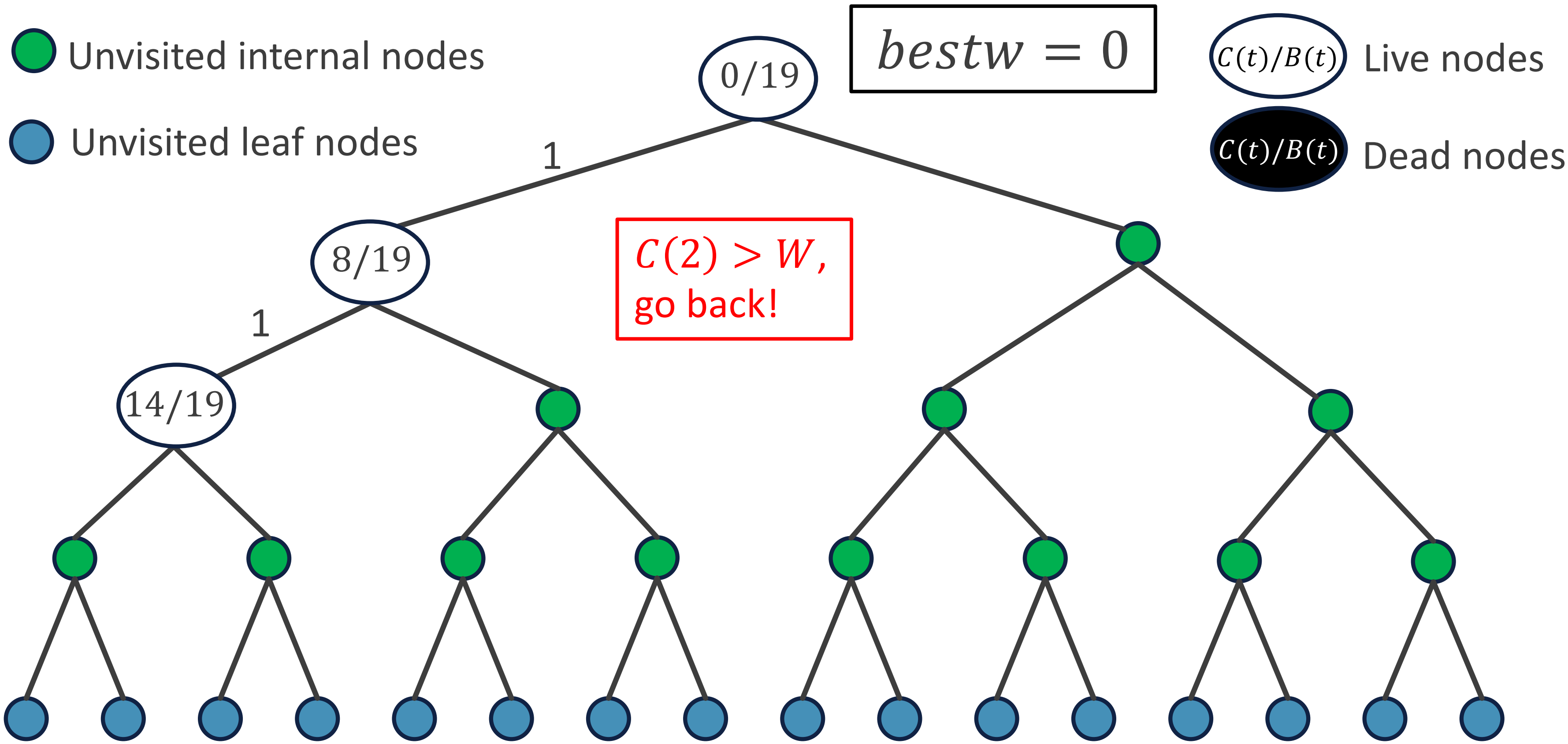
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



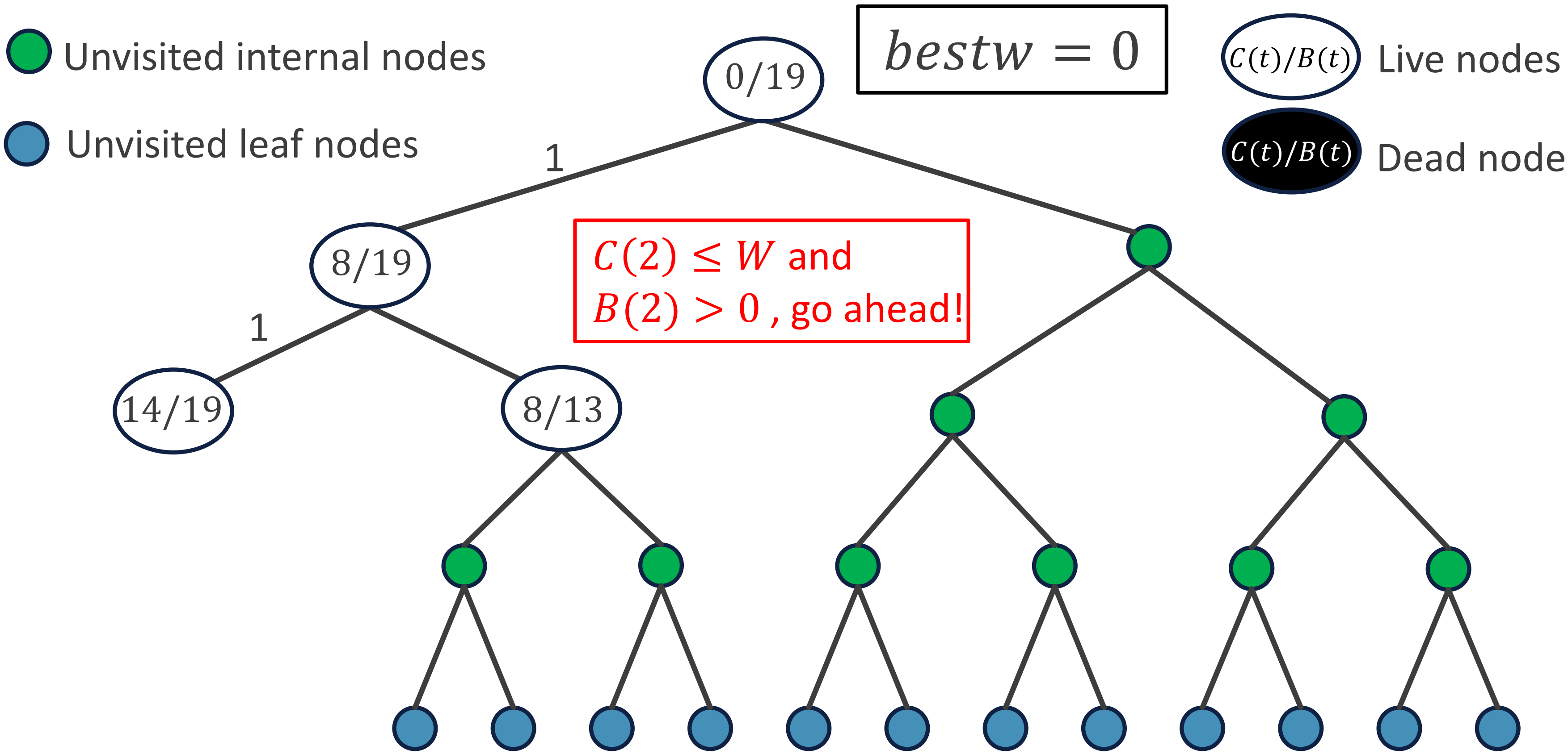
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



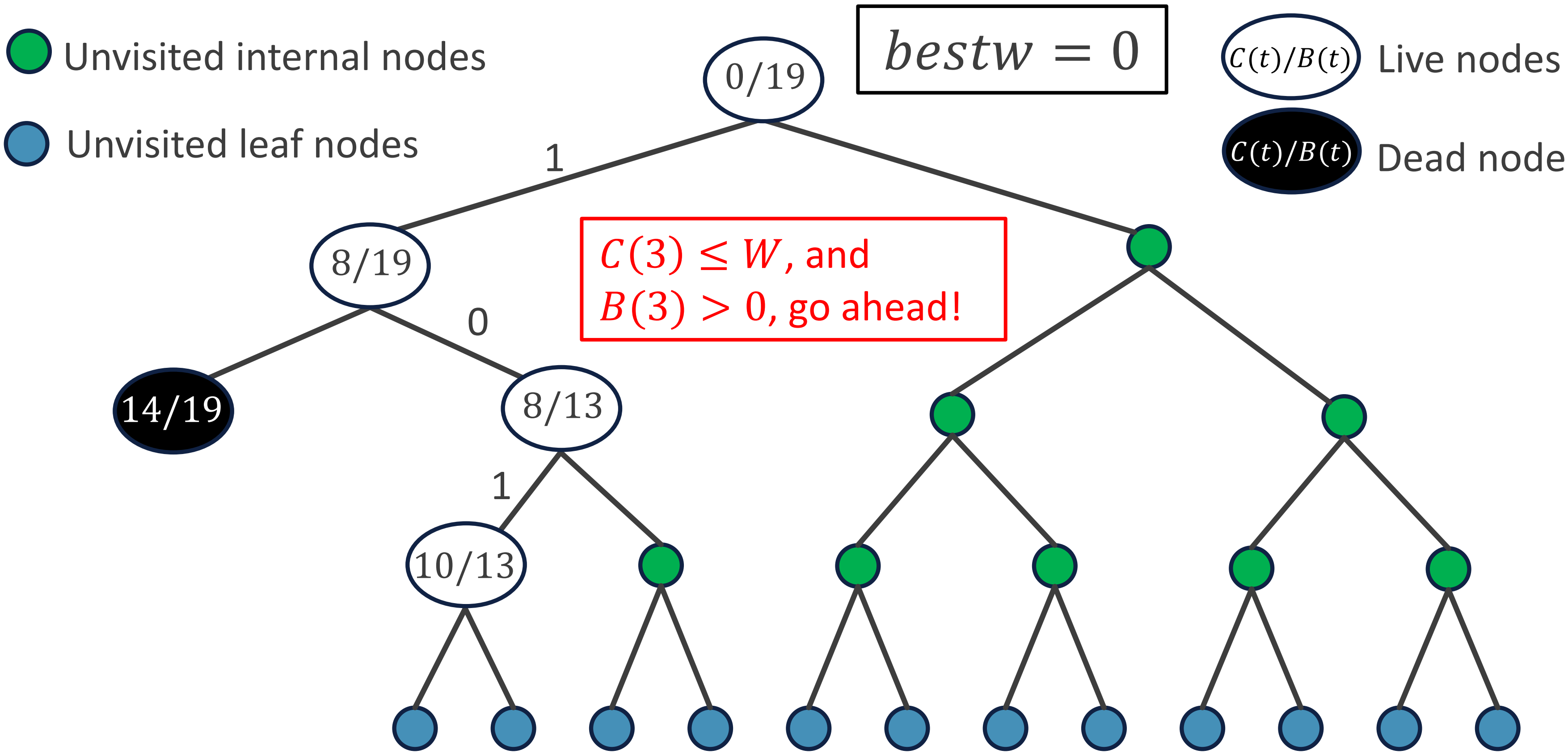
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



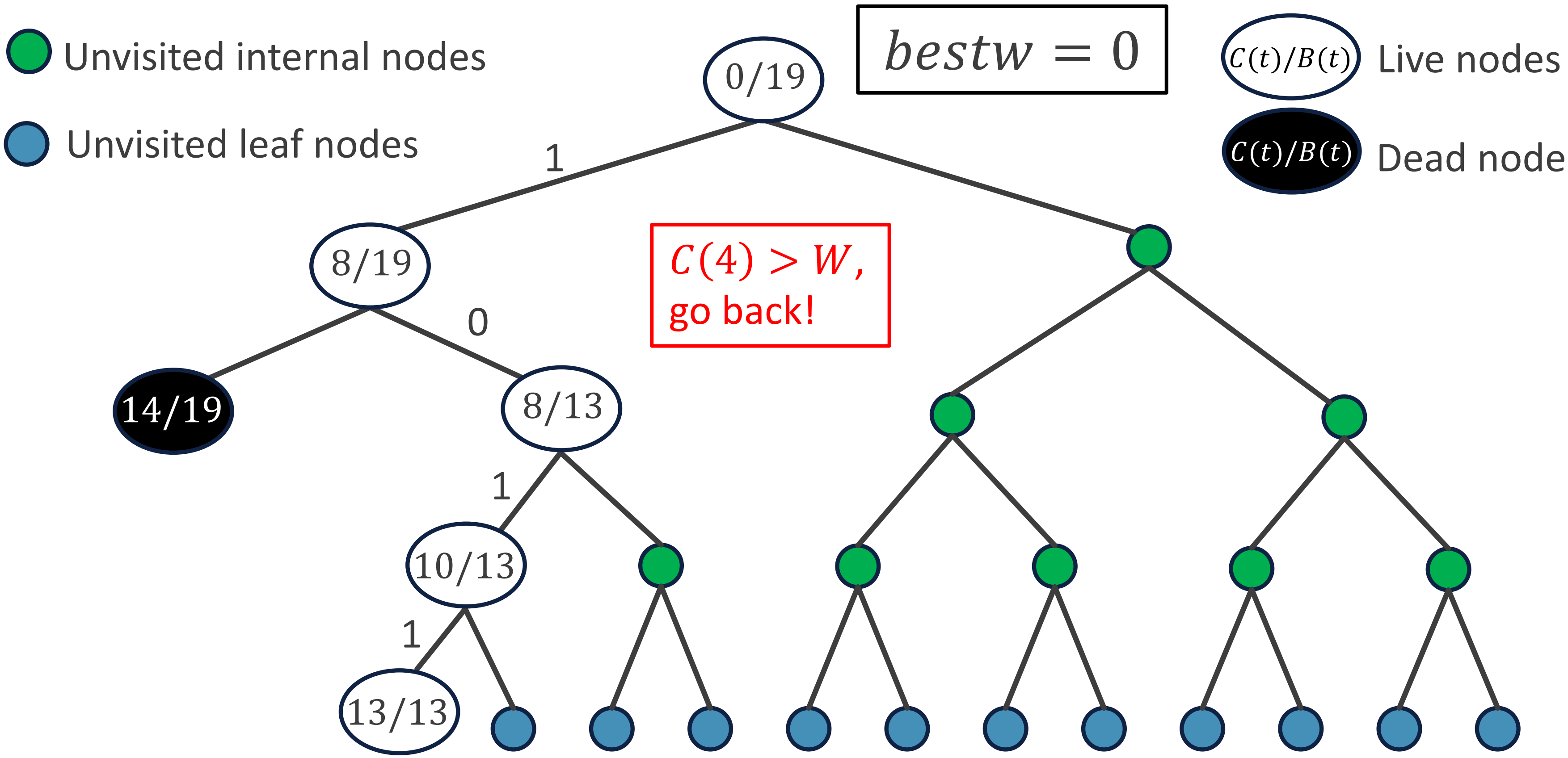
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



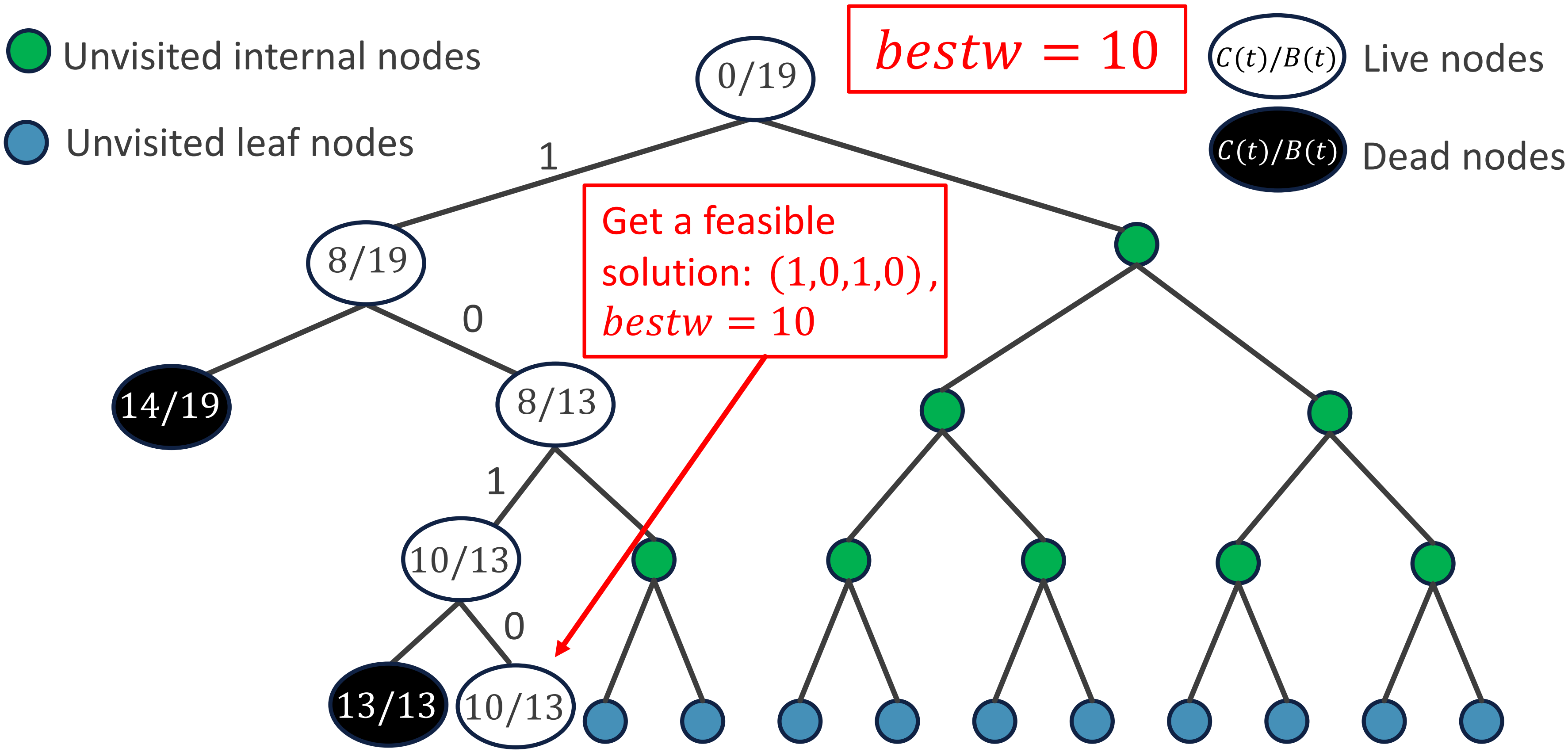
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



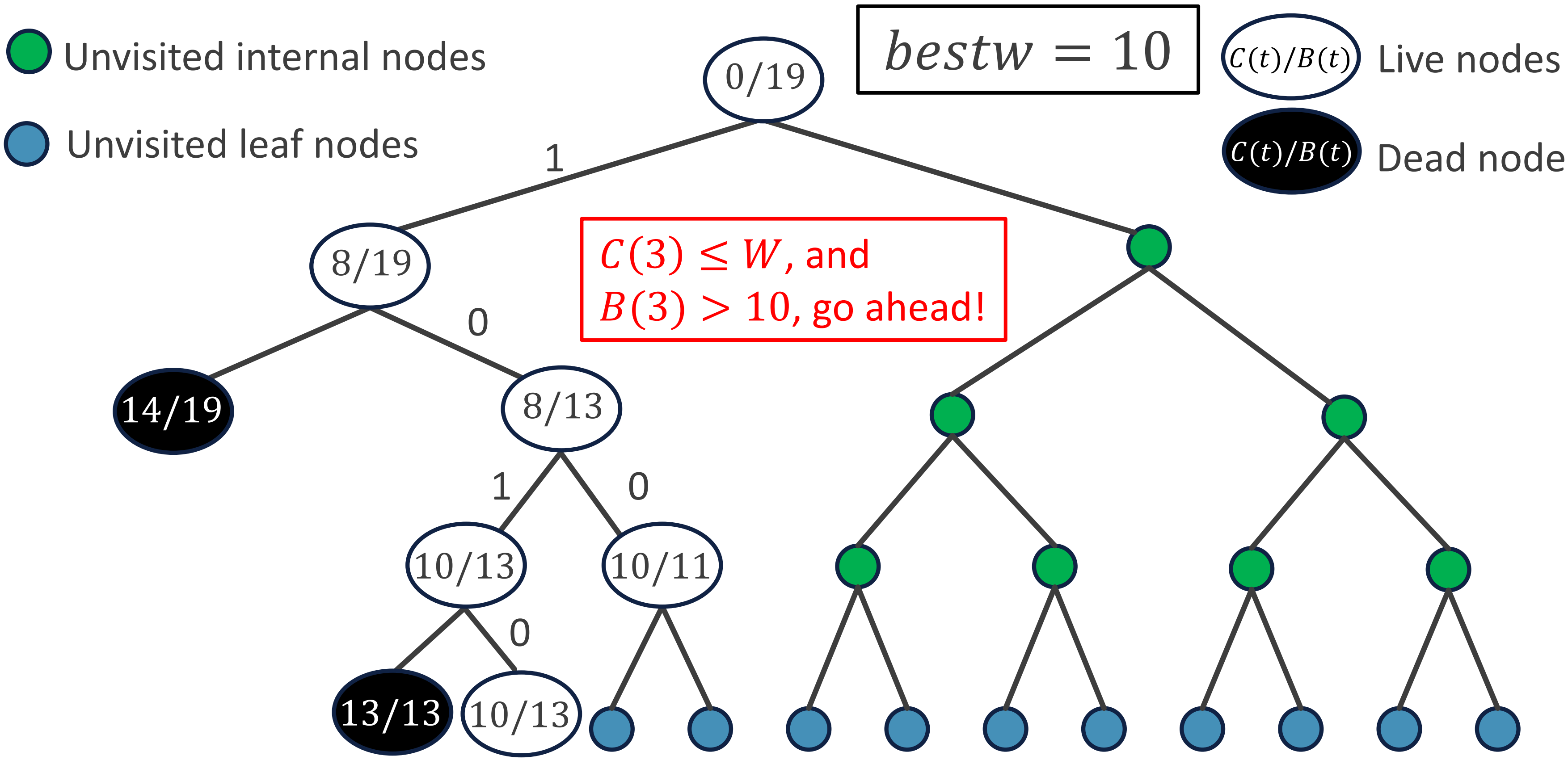
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



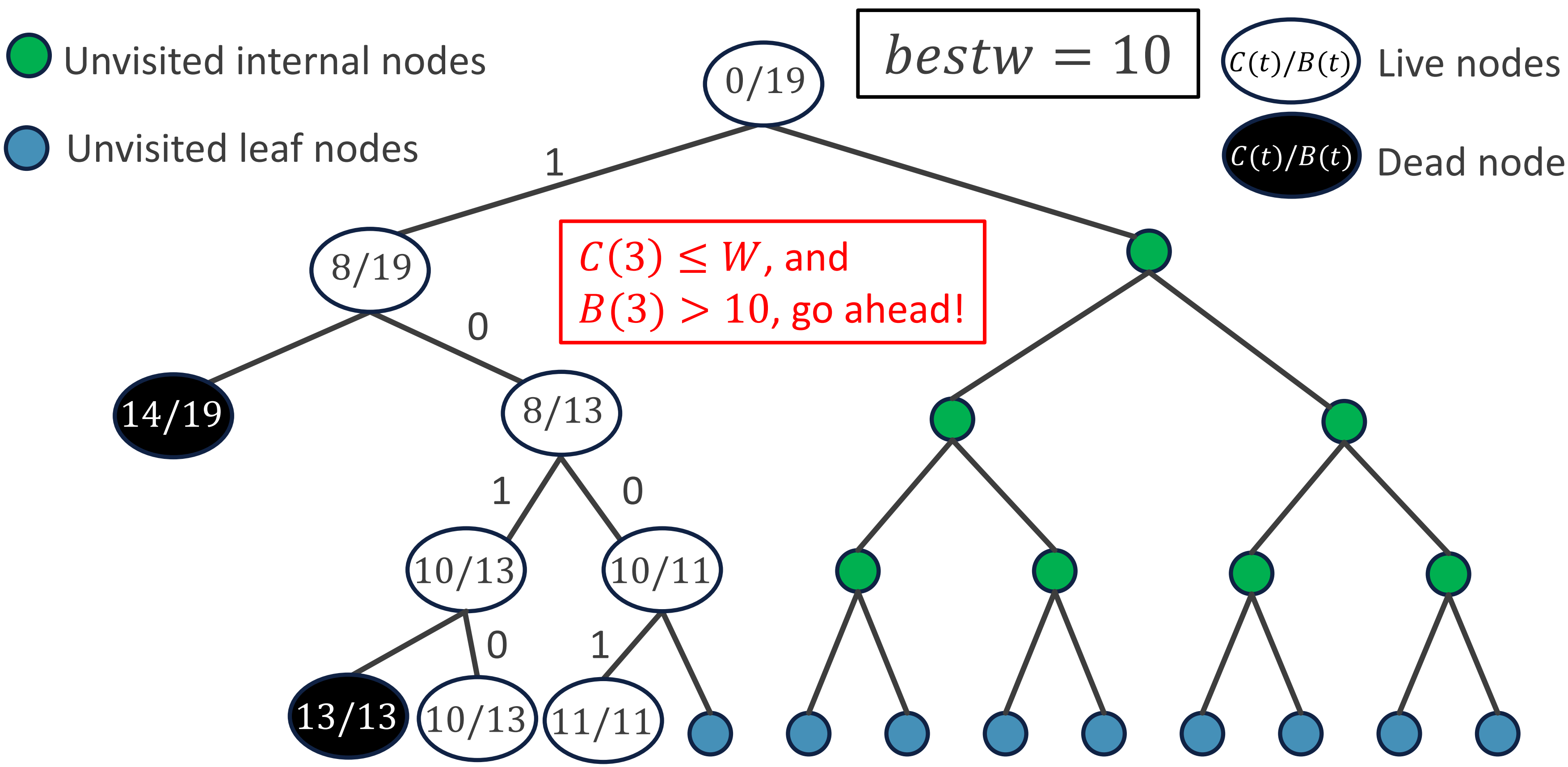
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



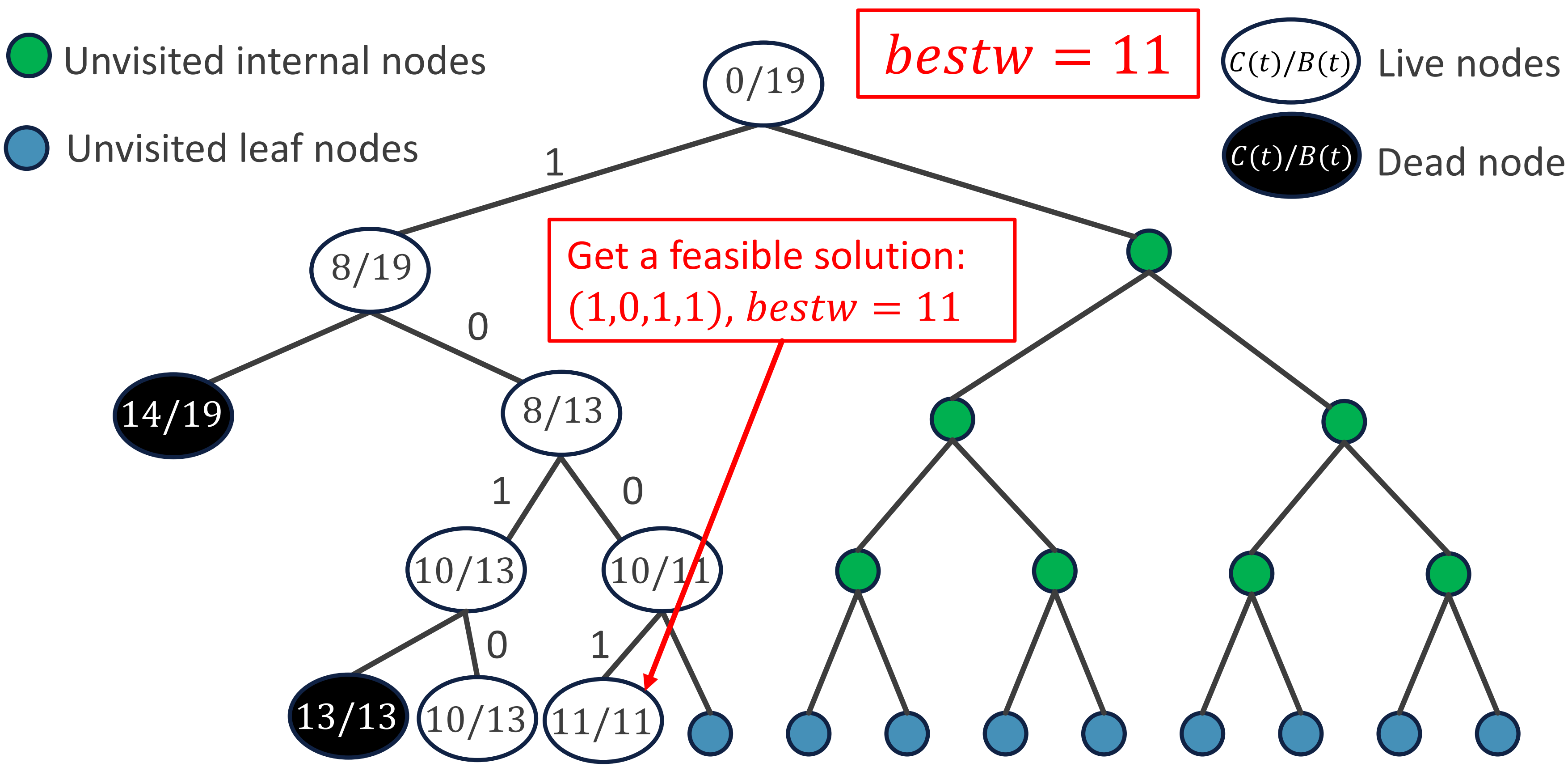
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



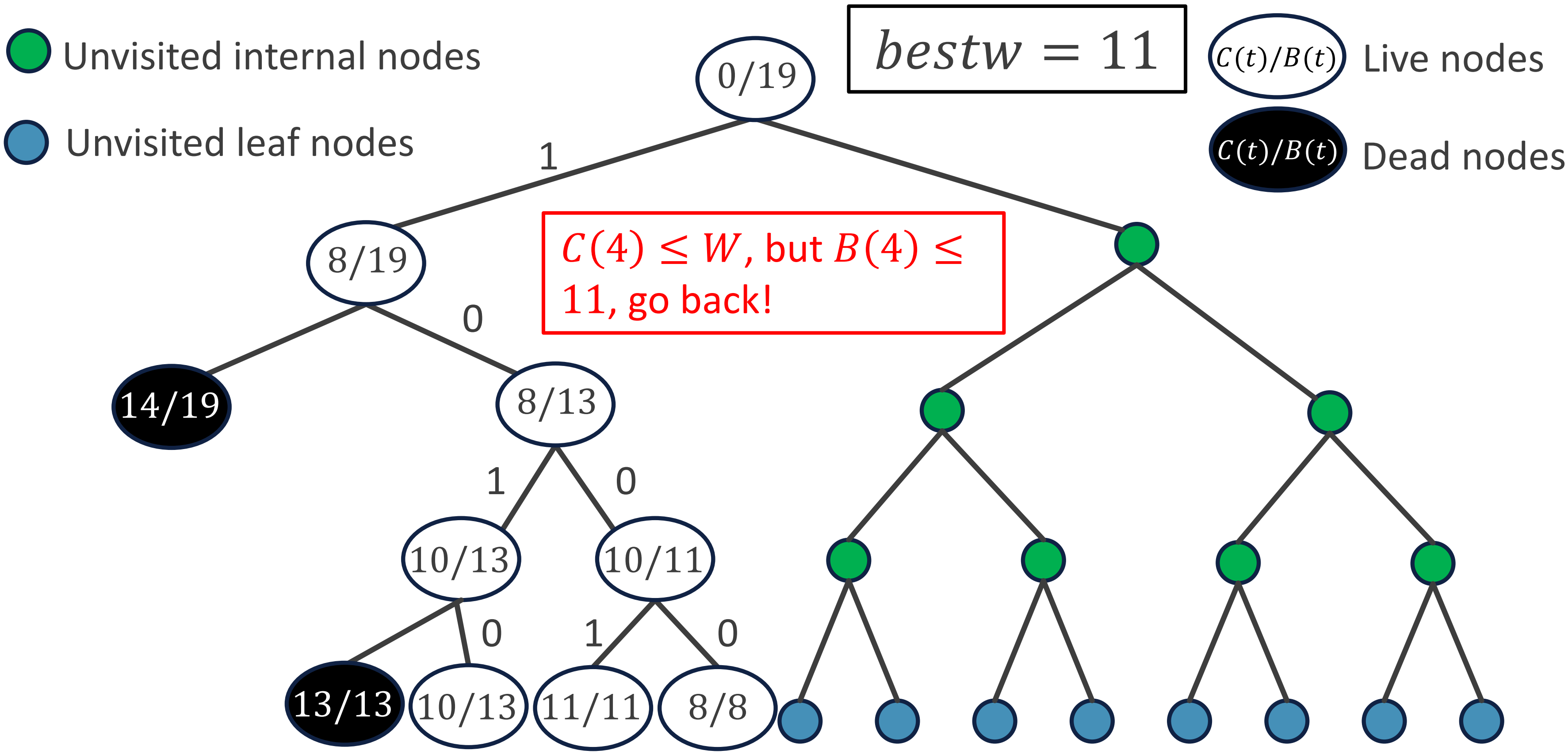
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



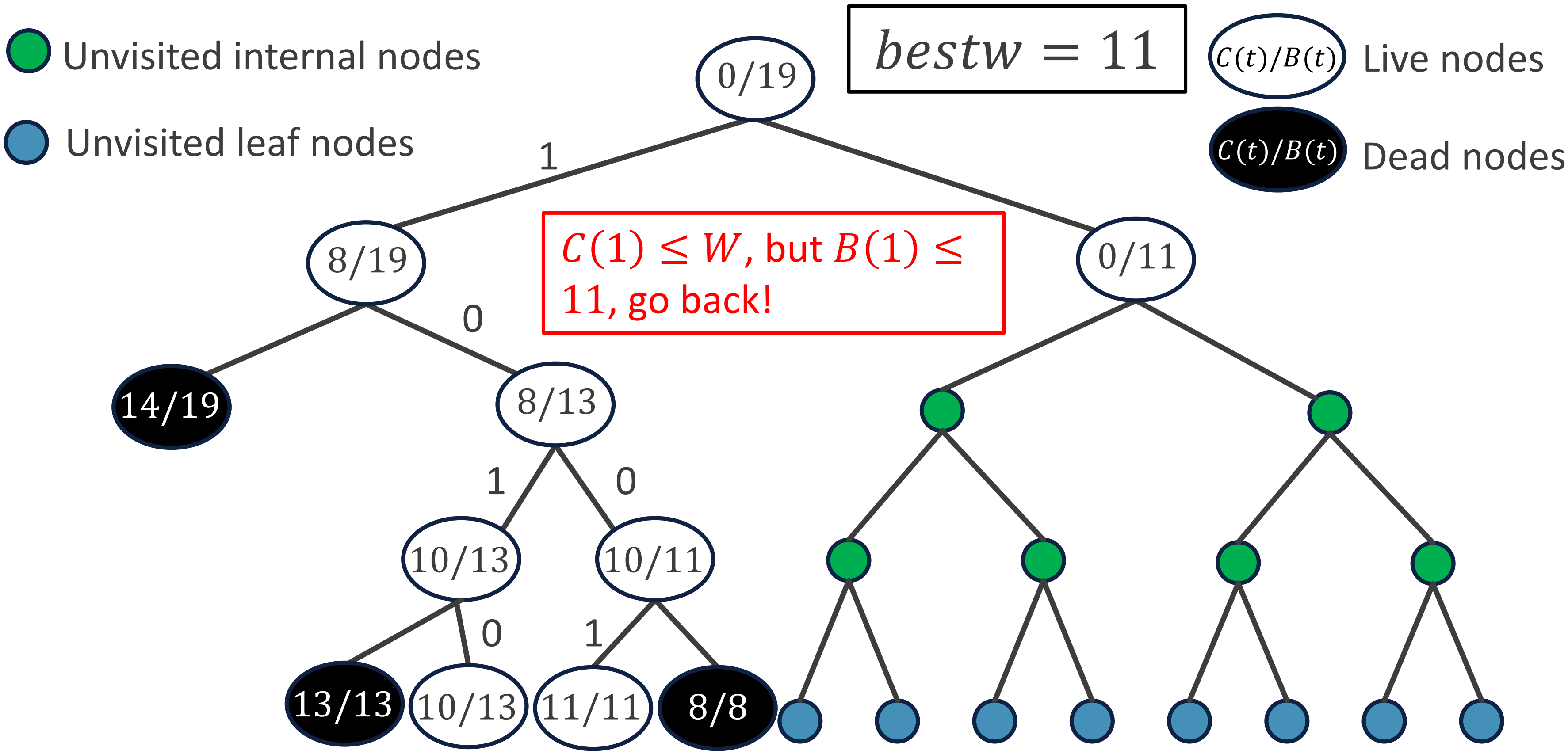
Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example



Backtracking for $n = 4, w = [8,6,2,3], W = 12$

Example



Backtracking for $n = 4$, $w = [8,6,2,3]$, $W = 12$

Example

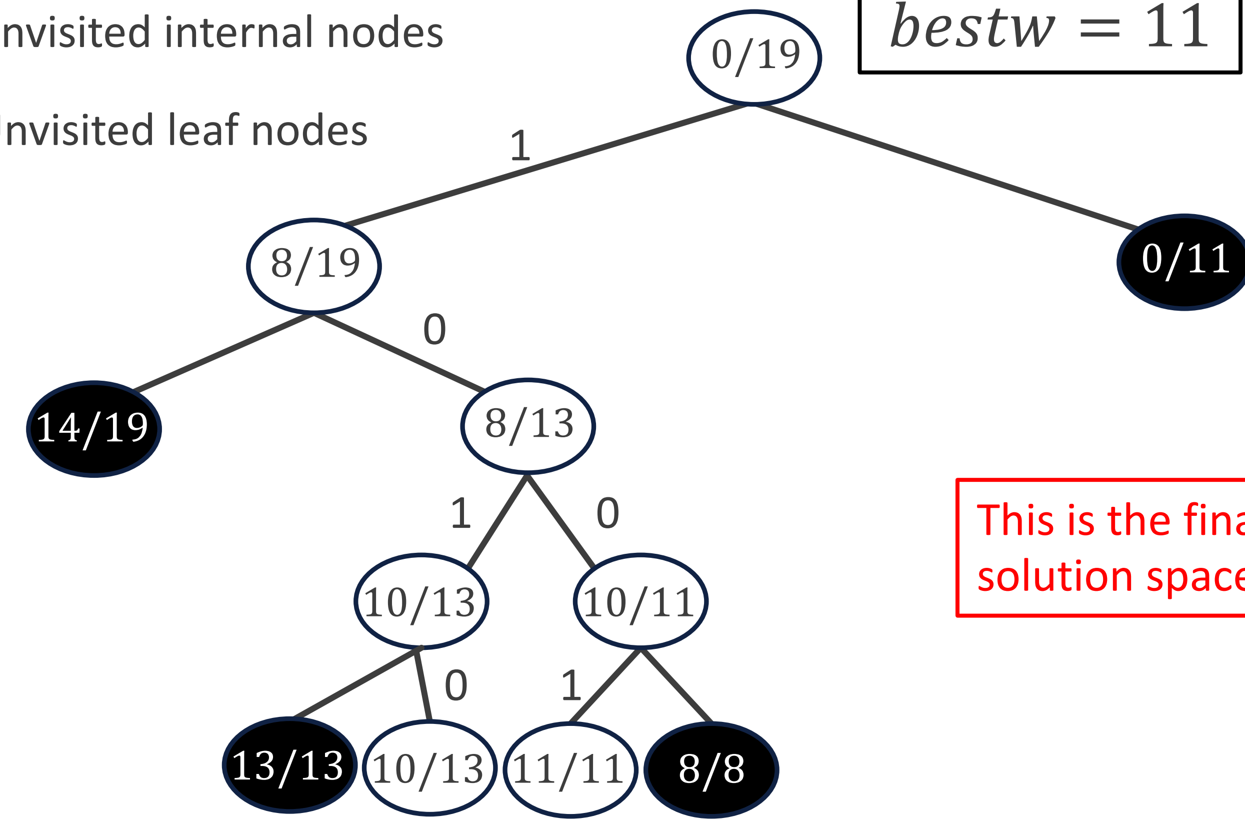
Unvisited internal nodes

Unvisited leaf nodes

$bestw = 11$

$C(t)/B(t)$ Live nodes

$C(t)/B(t)$ Dead nodes



This is the final pruned solution space tree.

Backtracking for $n = 4, w = [8,6,2,3], W = 12$

Pseudocode

```
ImprovedBacktrackLoading(i)
if i > n then
    if cw > bestw then
        bestw ← cw
        for j ← 1 to n do
            bestx[j] ← x[j]
else
    r ← r − w[i]
    if C(i) ≤ W then    x[i] ← 1
                        cw ← cw + w[i]
                        ImprovedBacktrackLoading(i + 1)
                        cw ← cw − w[i]
    if B(i) > bestw then  x[i] ← 0
                        ImprovedBacktrackLoading(i + 1)
    r ← r + w[i]
```

Record the best solution

Record the current solution

r is initialized as the total weight sum and reduced at the begging of each recursive call. After each recursive call, we add the weight back for going back.

Time Complexity

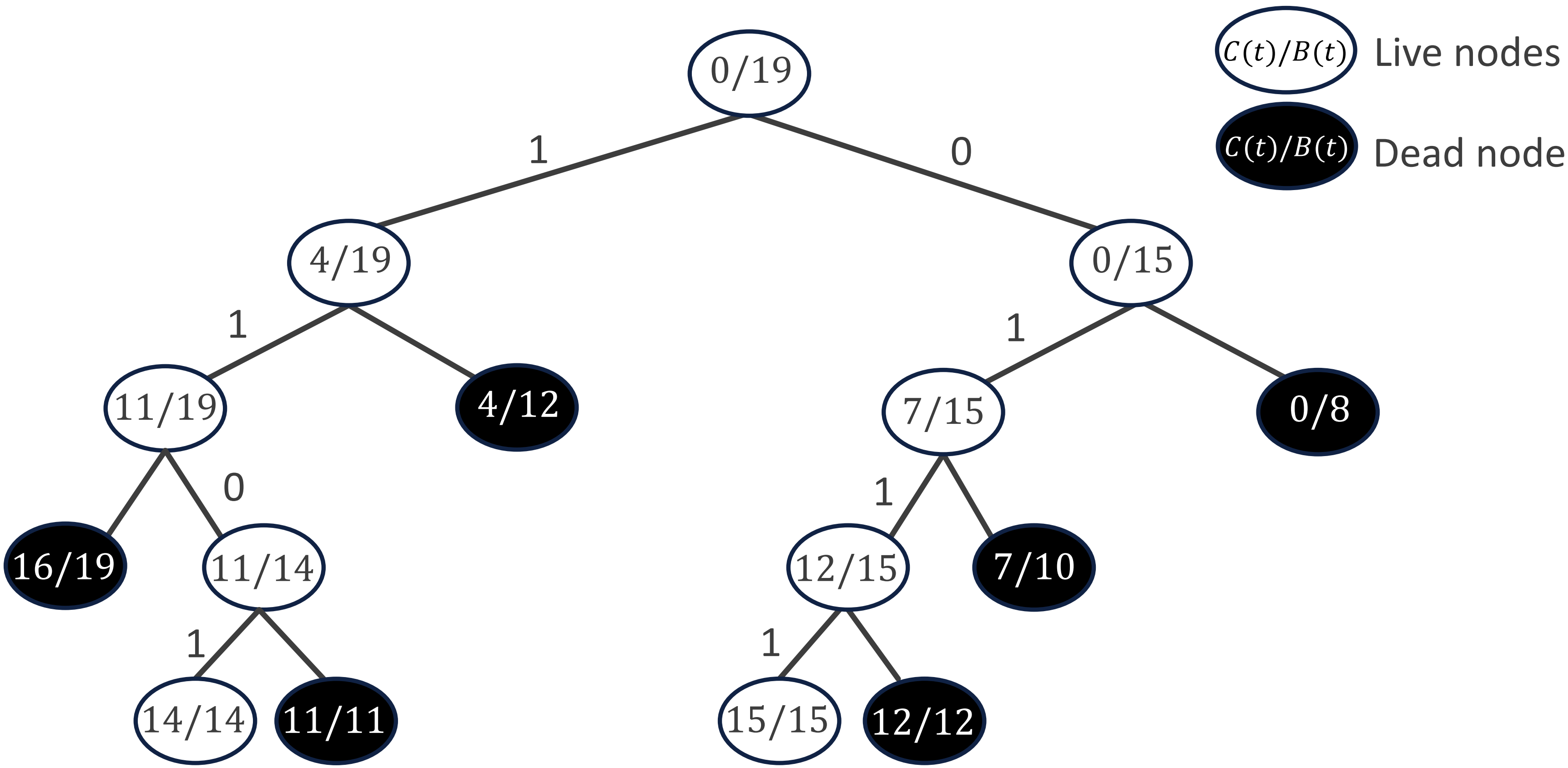
- Although backtracking seems very efficient. The time complexity for this algorithm is $O(n2^n)$.
  - 2^n is the time for searching the solution space.
  - n is the time to store the best solution.
- This is a sad story…

Classroom Exercise

- Draw the pruned solution space tree for the following container loading problem instance.

$$n = 4, w = [4,7,5,3], W = 15$$

Classroom Exercise



Backtracking for $n = 4, w = [4,7,5,3], W = 15$

Classroom Exercise

- In the Sum-of-Subsets problem, there are n positive integers (weights) w_i and a positive integer W.
- The goal is to find all subsets of the integers that sum to W.
- Example:
  - Suppose that $n = 4$, $W = 13$, and $w = [3,4,5,6]$.
  - The solutions is $[1,1,0,1]$ because $w_1 + w_2 + w_4 = 3 + 4 + 6 = 13$,
- Design the constraint function and bounding function, and the corresponding condition.
- Draw the pruned solution space tree of the above example.

Classroom Exercise

- The constraint function $C(i)$ and its condition are same as the container loading problem:

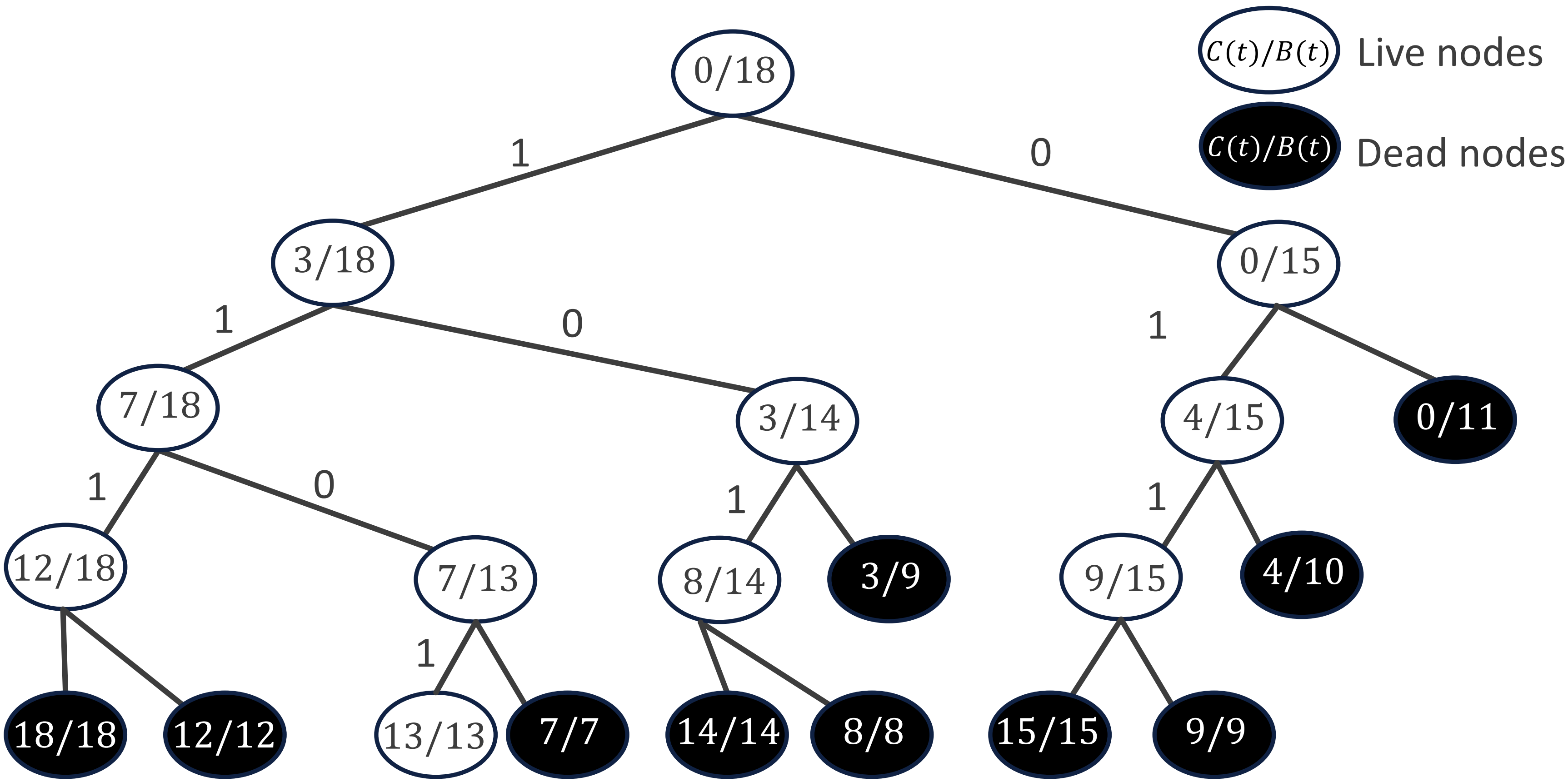
$$C(i) > W$$

- The bounding function $B(i)$ is same as the container loading problem, but the condition is different:

$$B(i) < W$$

  - Instead of comparing with $bestw$ in the container loading problem.

Classroom Exercise



Backtracking for $n = 4$, $w = [3,4,5,6]$, $W = 13$

0/1 KNAPSACK PROBLEM

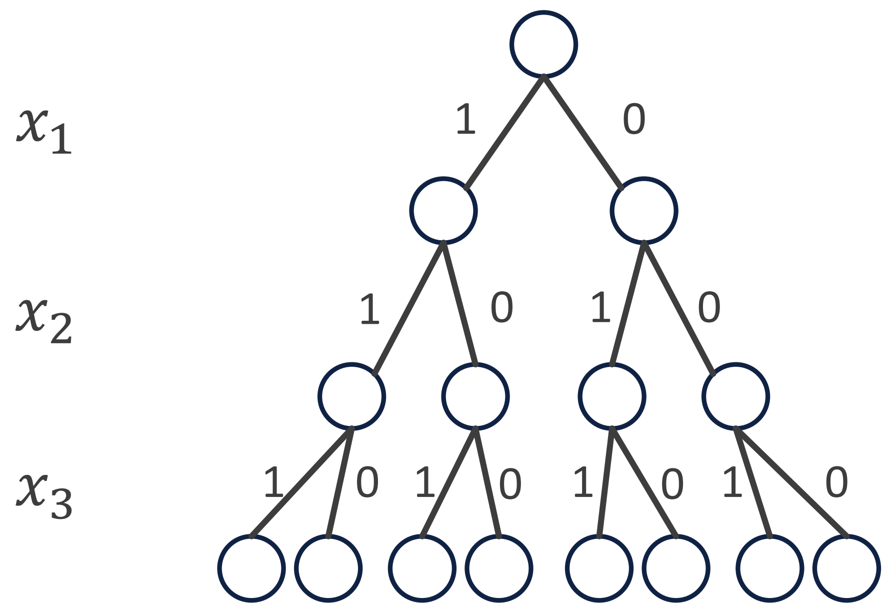
0/1 Knapsack Problem

- There are n items: the ith item is worth v_i dollars and weights w_i kg. The capacity of knapsack is W kg.
- Assuming that the solutions are represented by vectors $(x_1, x_2, \dots, x_n)$, where $x_i \in \{0,1\}$. 1 denotes taking item i and 0 denotes not taking item i.
- The 0/1 knapsack problem can be formally stated as follows:

$$\max \sum_{i=1}^{n} v_i x_i \qquad s.t. \sum_{i=1}^{n} w_i x_i \le W$$

0/1 Knapsack Problem

- It is nothing but a high-level container loading problem.
- The size of solution space and the solution space tree are exactly same as the container loading problem.



Solution space tree with $n = 3$

0/1 Knapsack Problem

- Constraint function: also exactly same as the container loading problem!
- Let $cw(i)$ denote the current weight up to level i, namely

$$cw(i) = \sum_{j=1}^{i} w_j x_j$$

then the constraint function is

$$C(i) = cw(i-1) + w_i$$

- The pruning condition is $C(i) > W$, which means there is no capacity to take container i.

0/1 Knapsack Problem

- The bounding function:

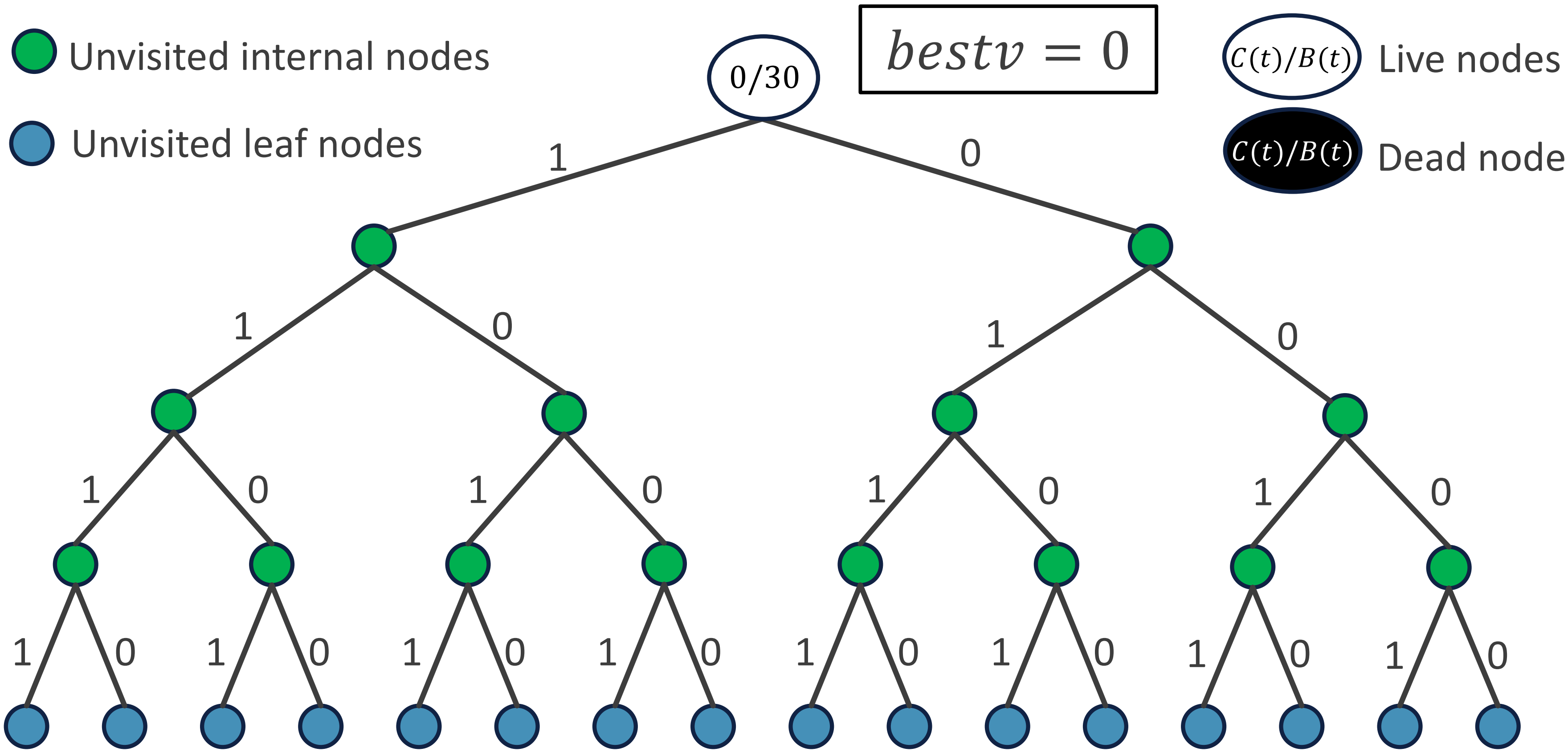
$$B(i) = C(i) + r(i)$$

However, different from the bounding function in the container loading problem, $r(i)$ denotes the value sum of the remaining items, namely,

$$r(i) = \sum_{j=i+1}^{n} v_j$$

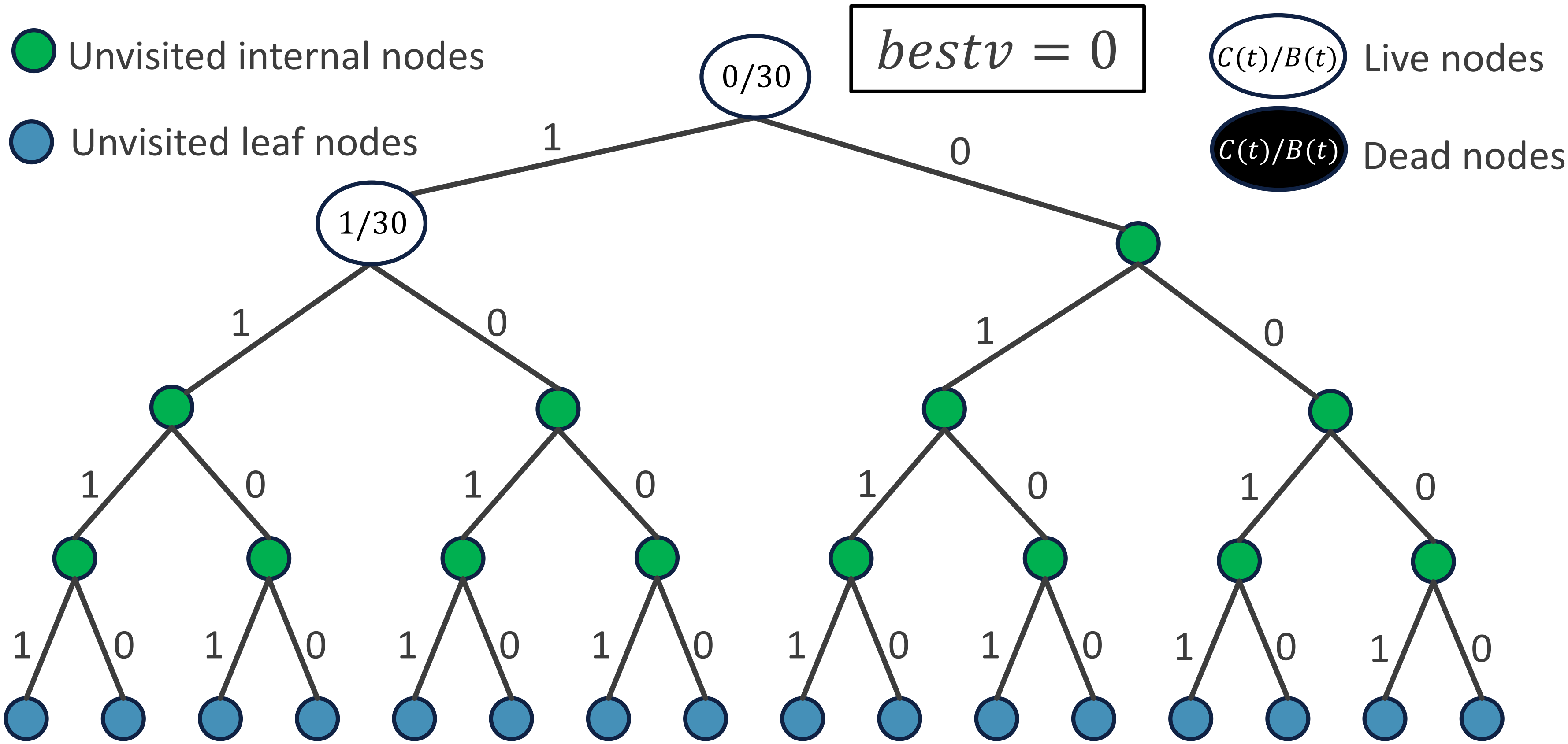
- The pruning condition is $B(i) \leq bestv$, which means the continuing searching along this branch will not give better solution.

Example



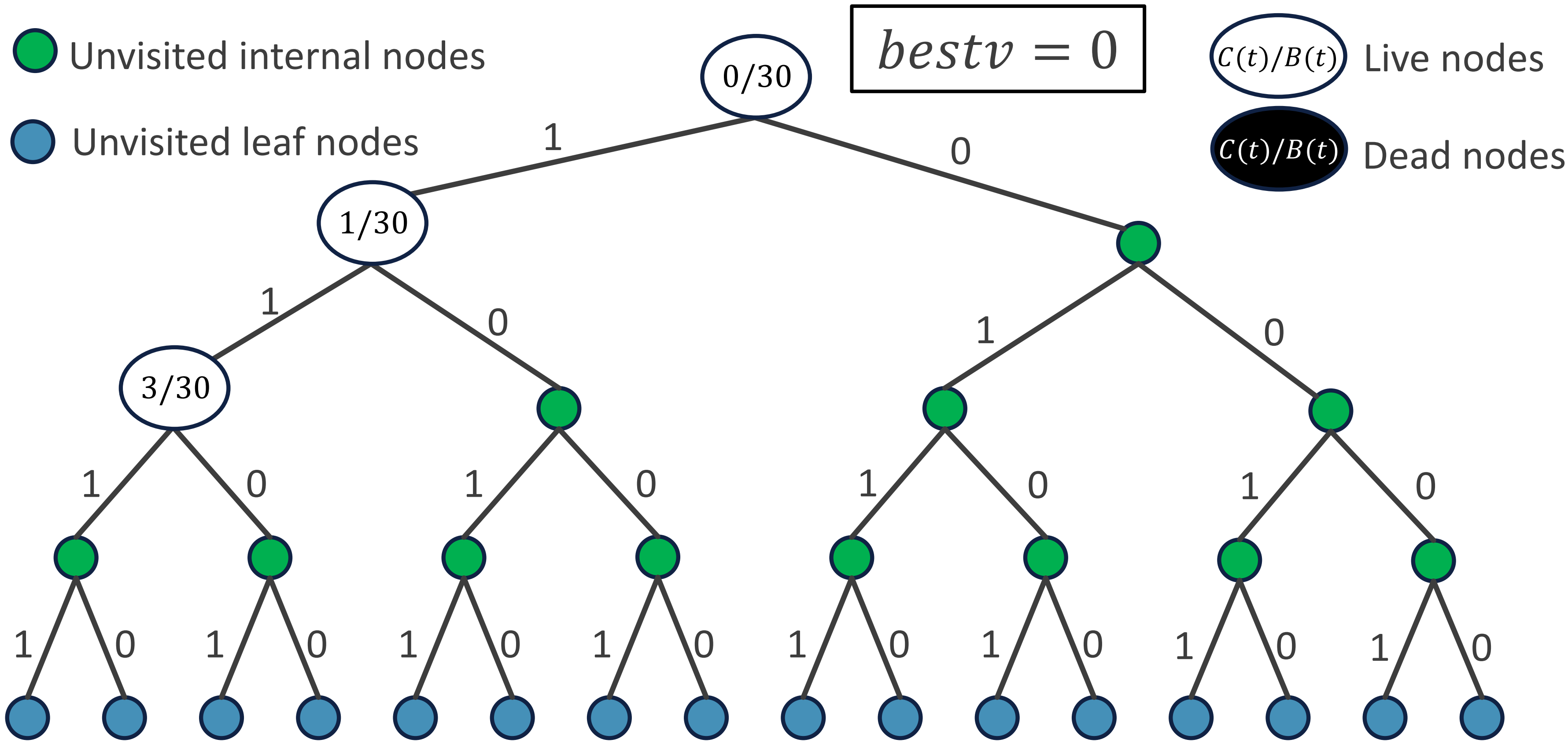
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



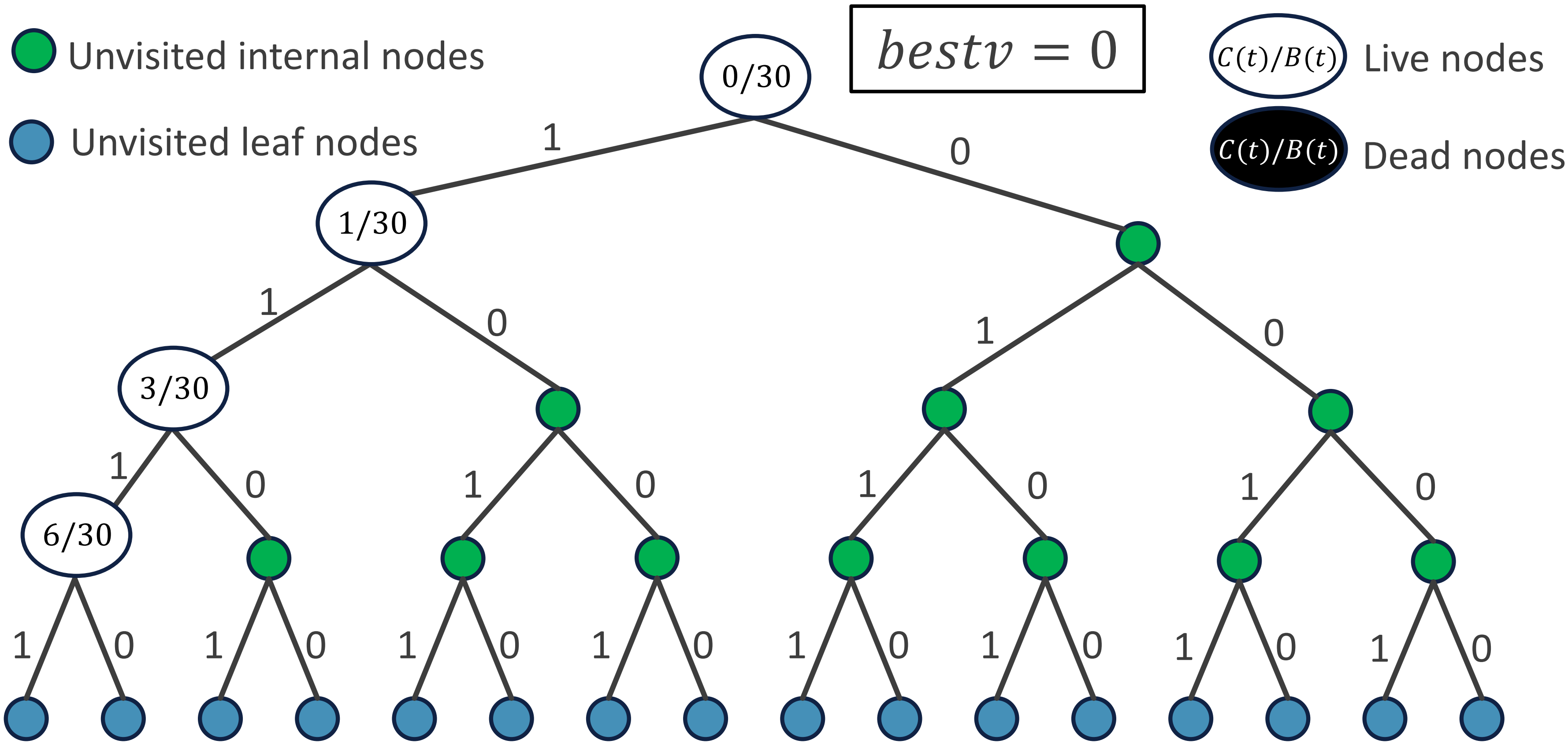
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

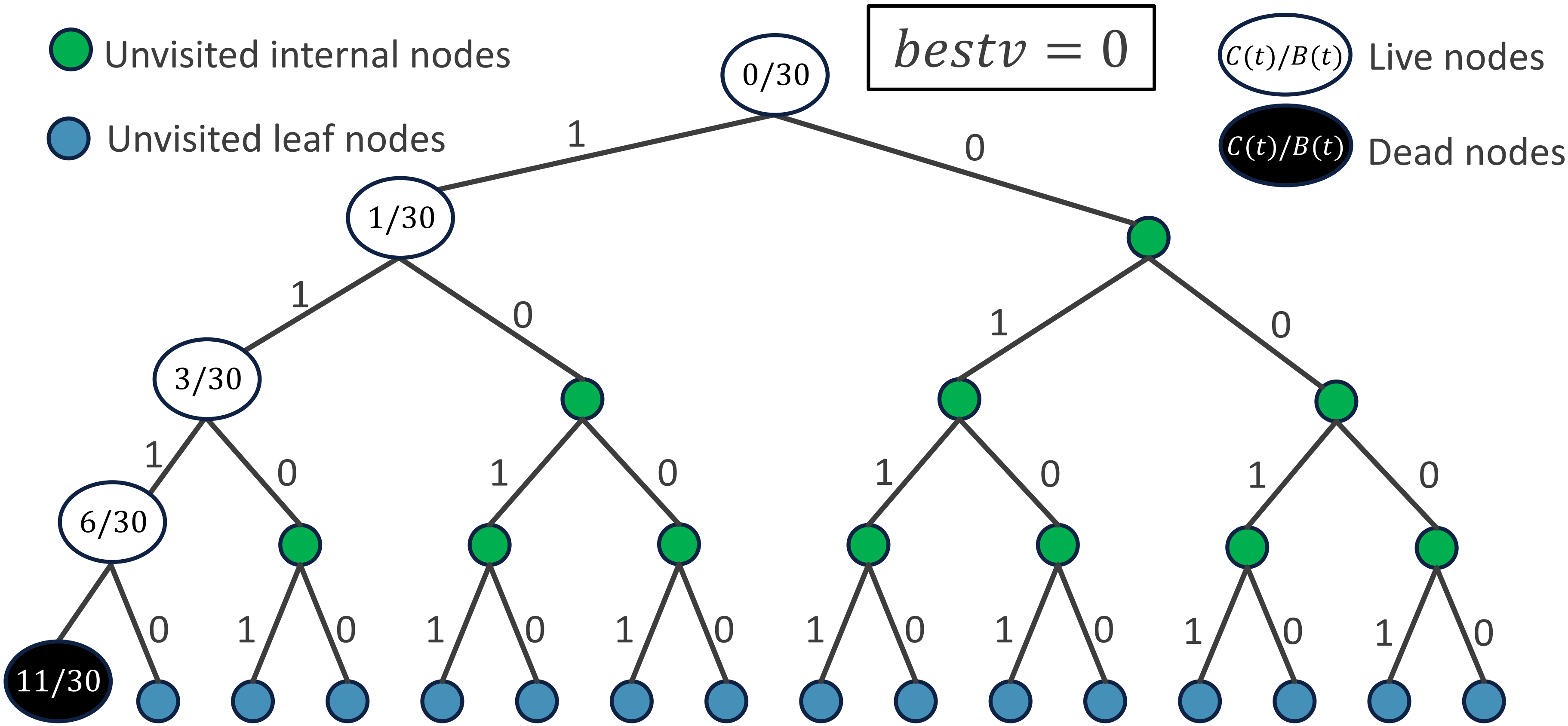
Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

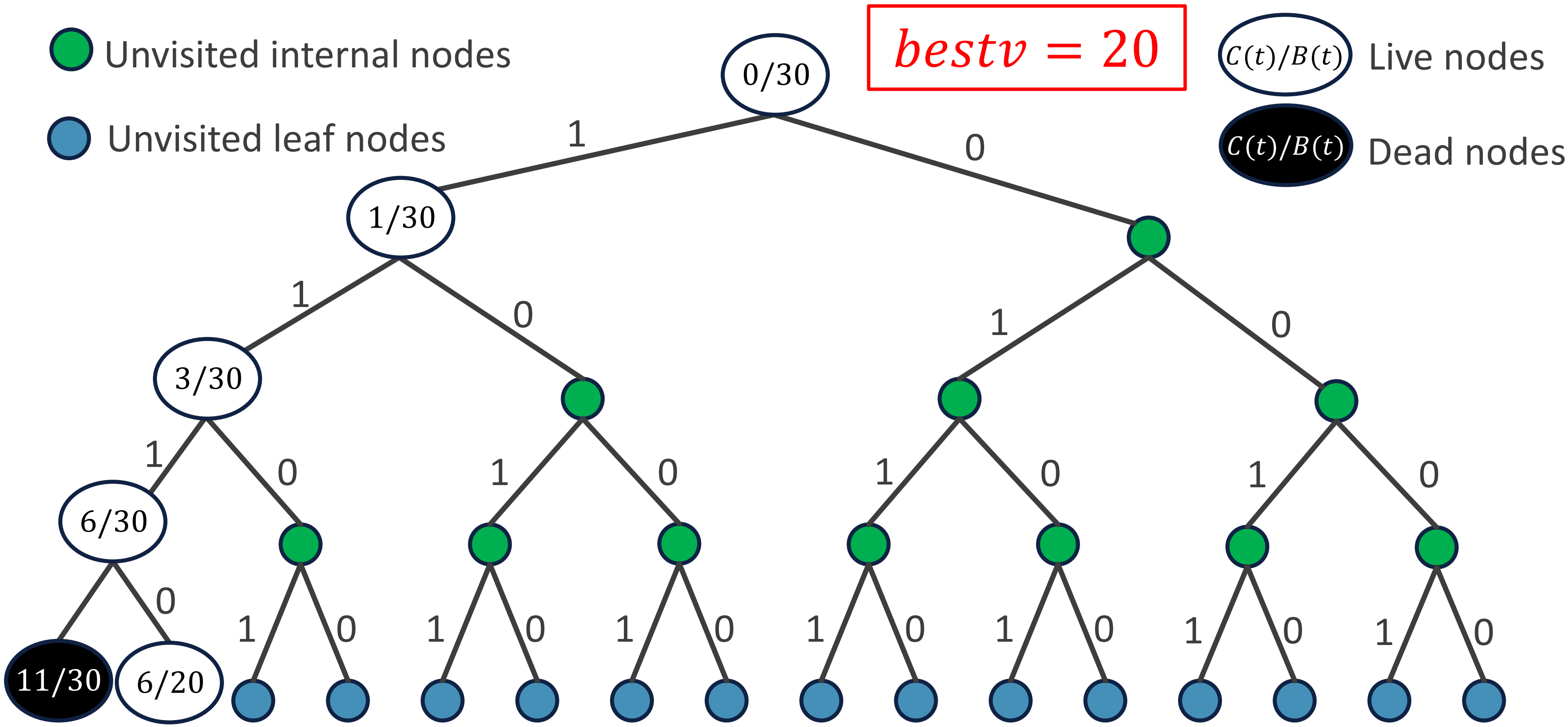
厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Example



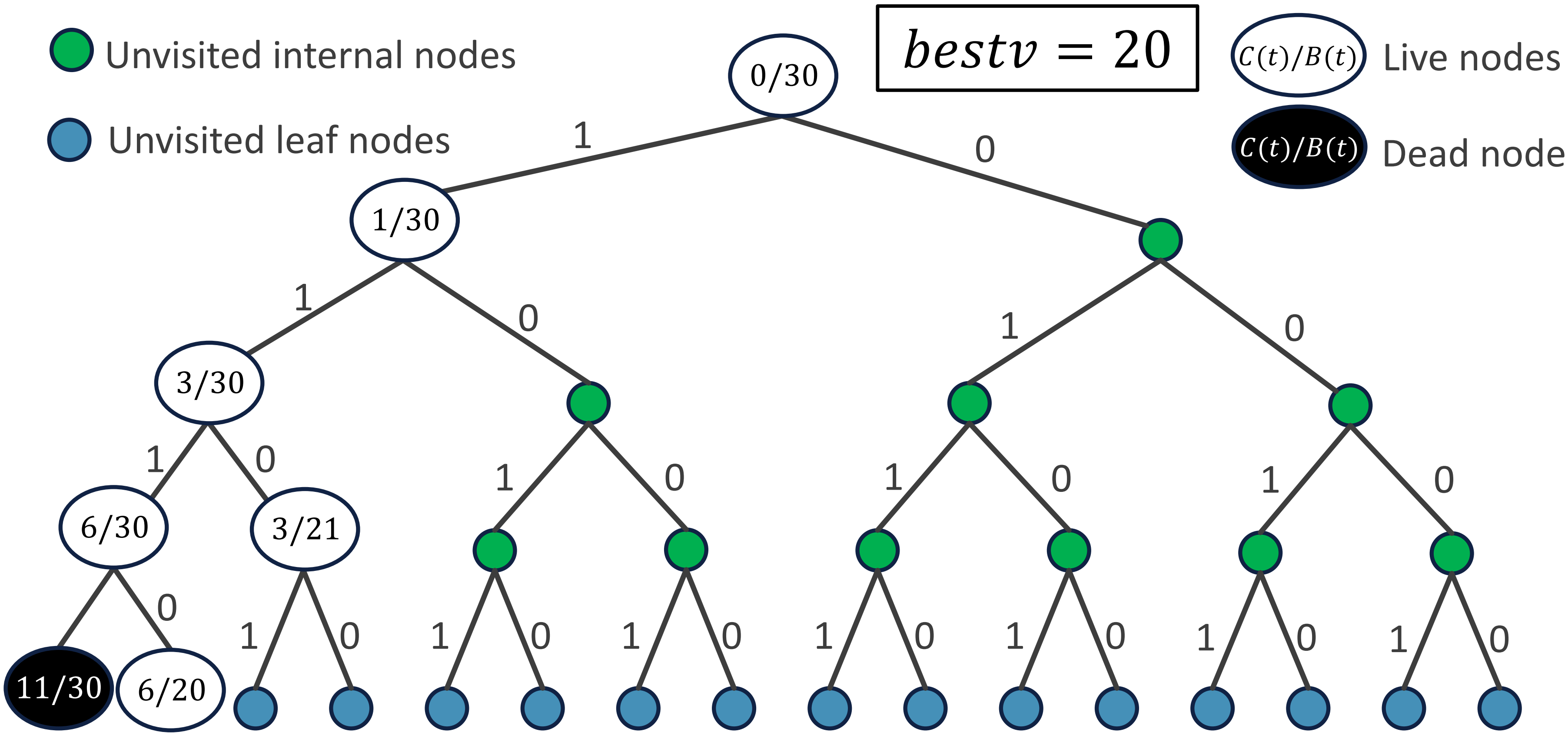
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



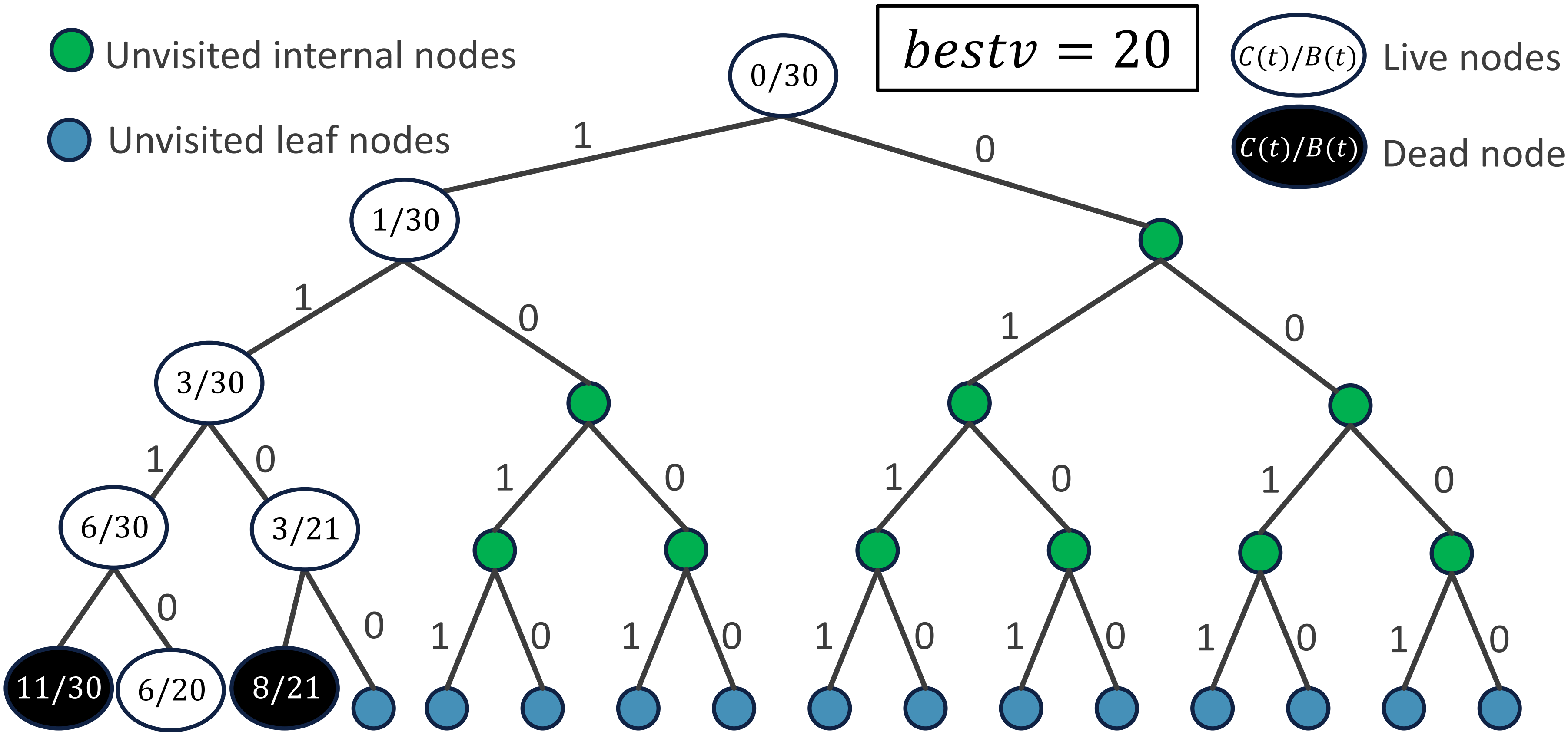
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



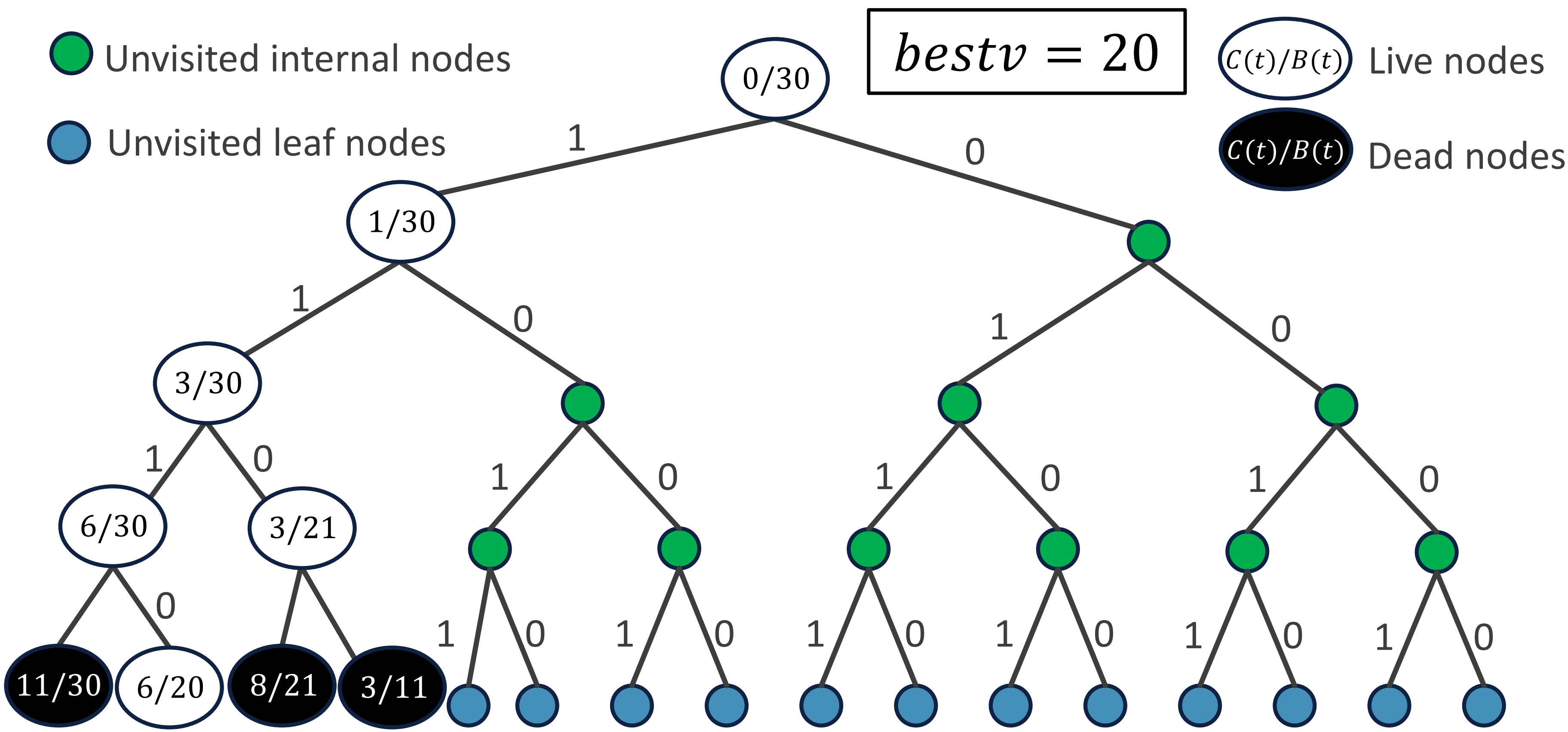
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



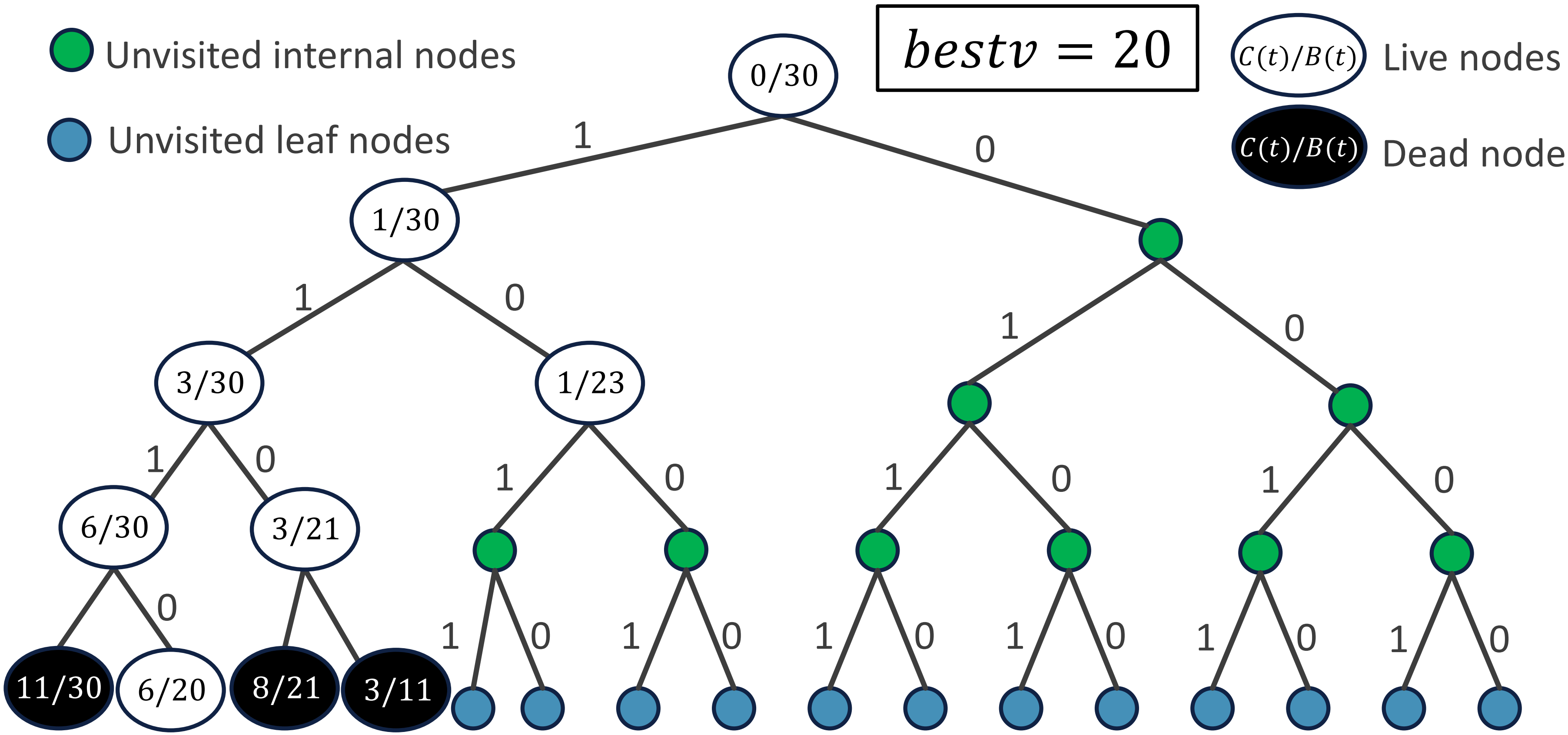
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



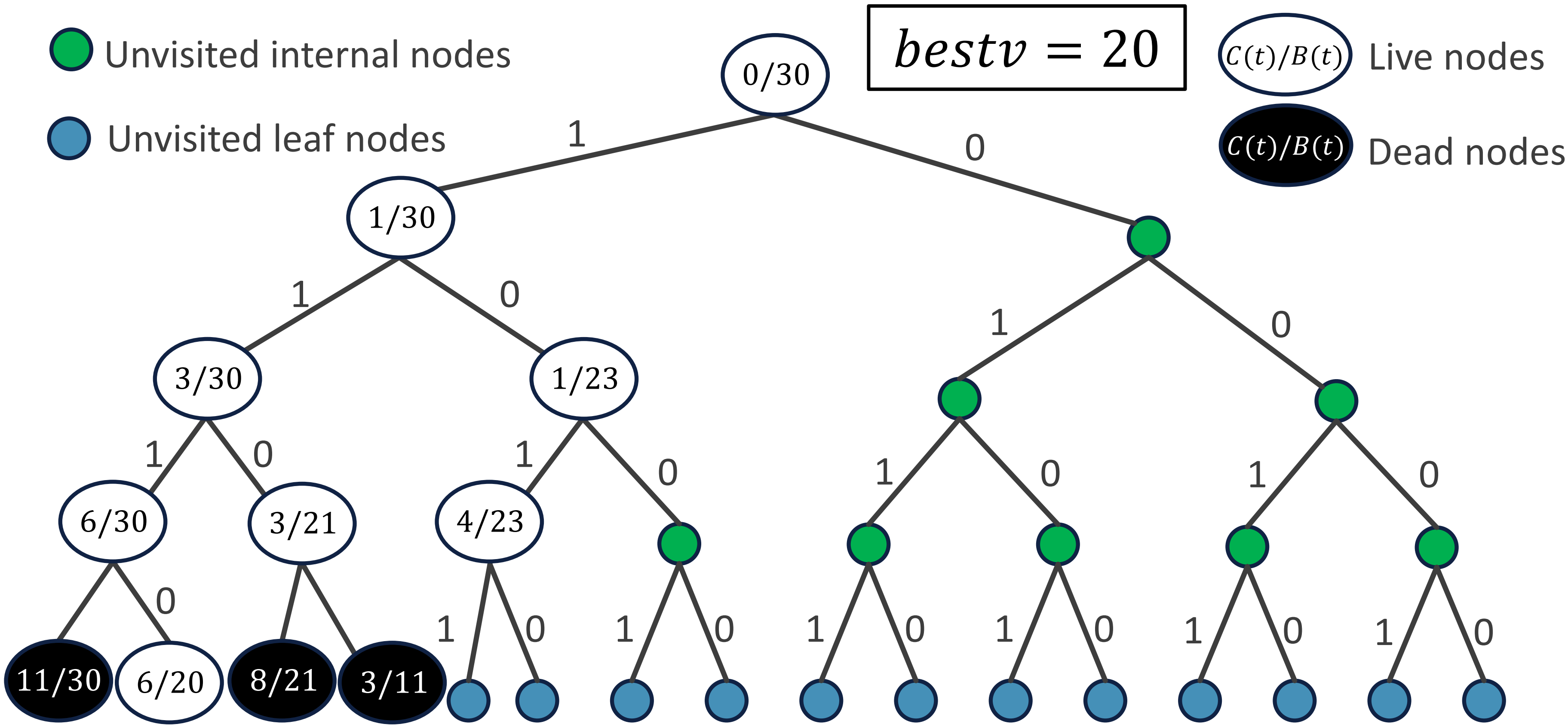
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



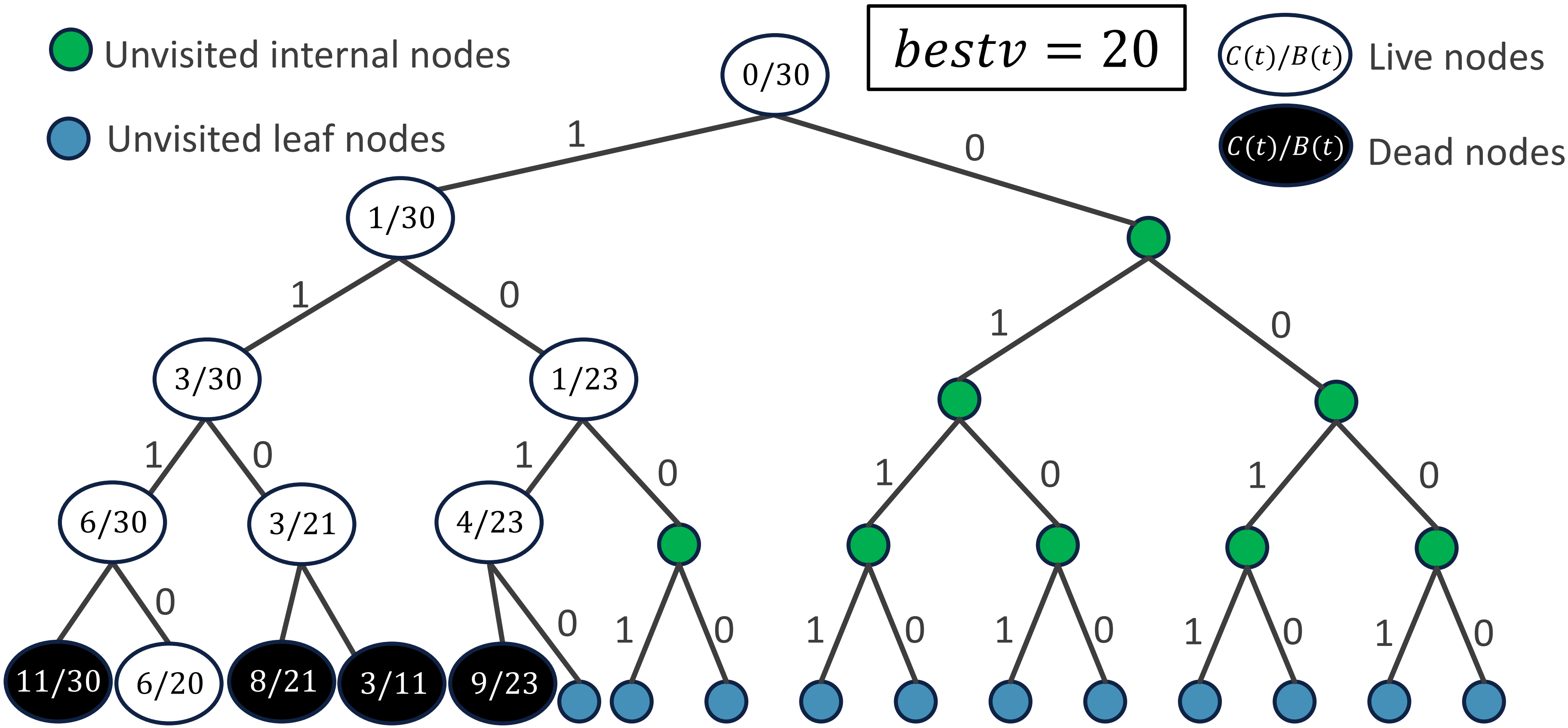
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



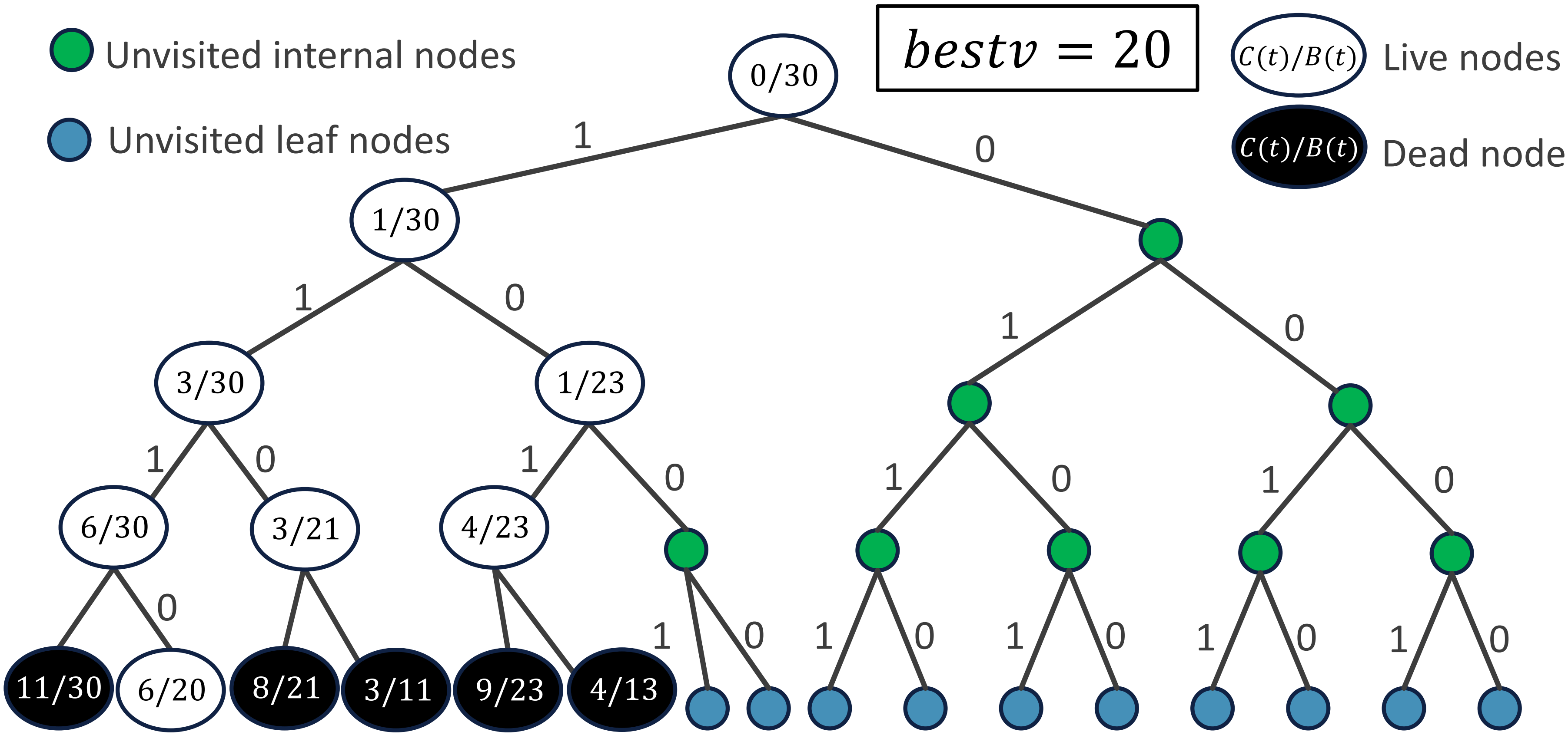
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



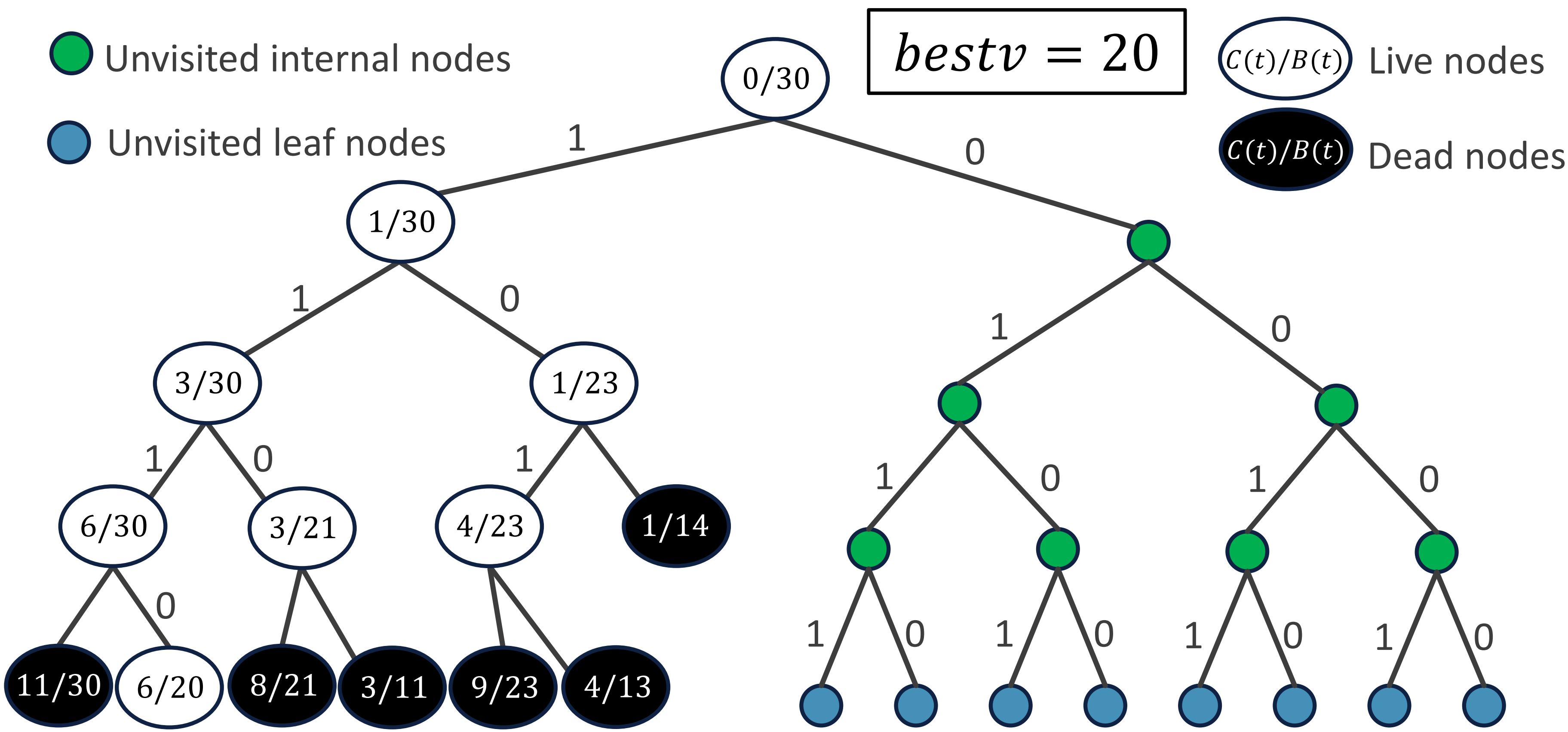
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



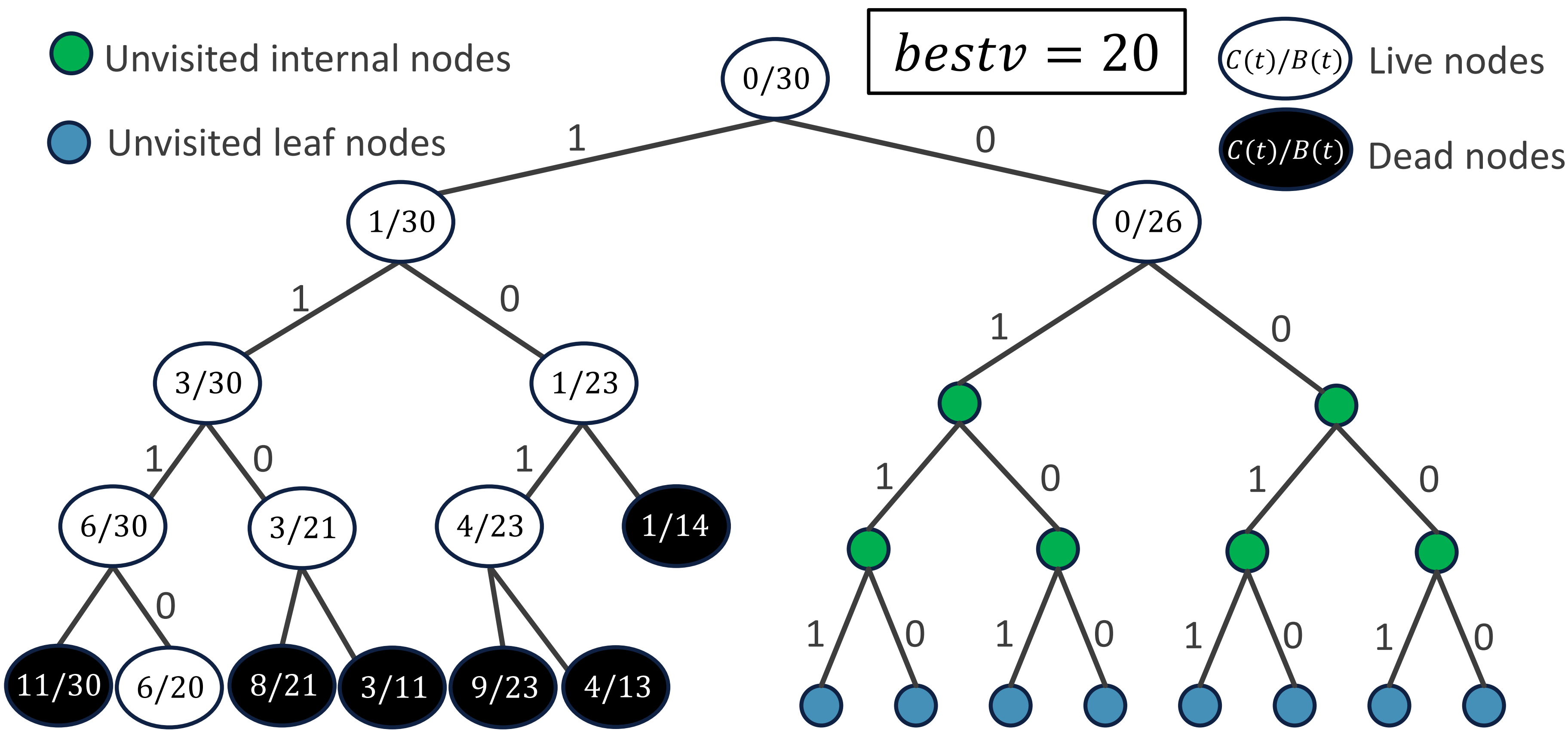
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



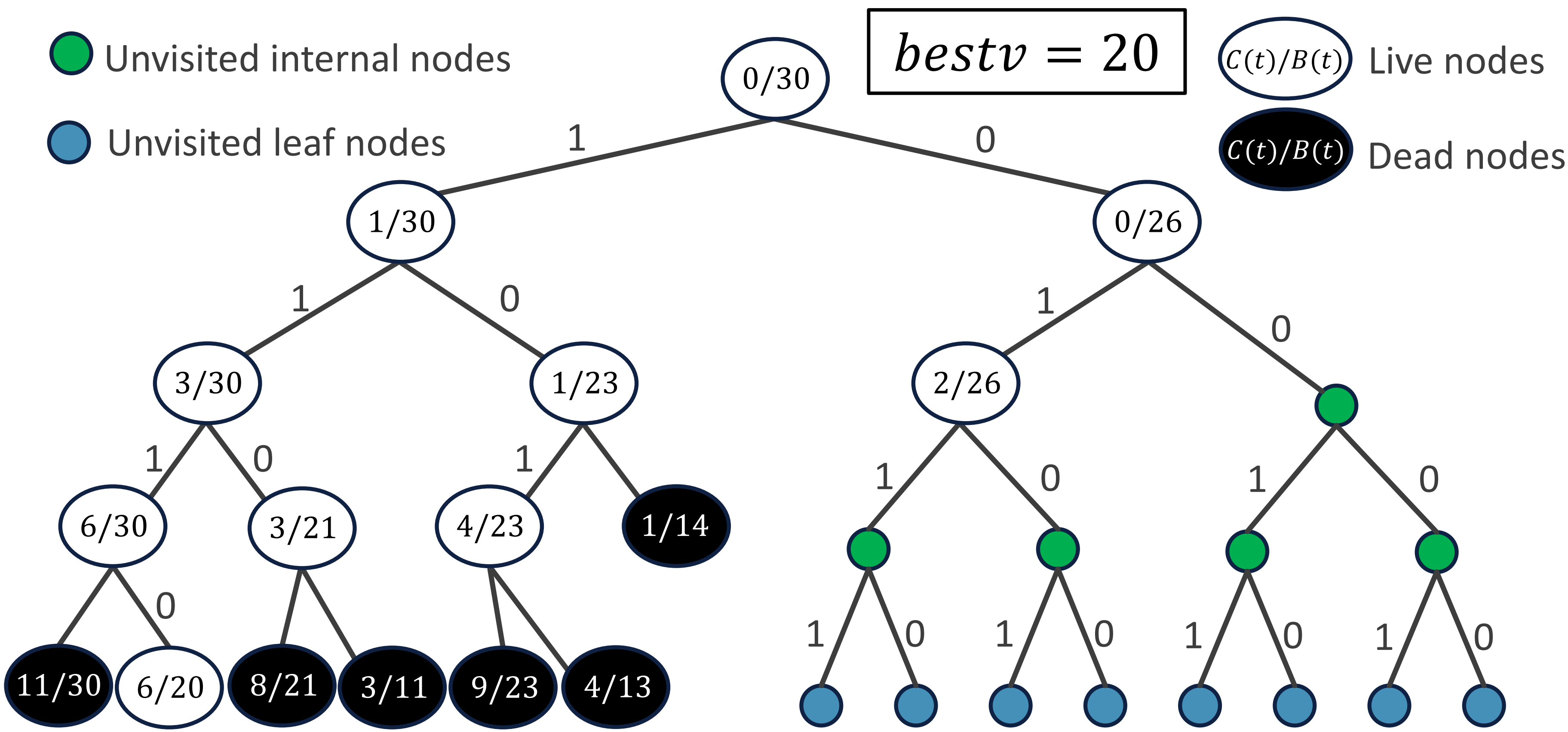
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



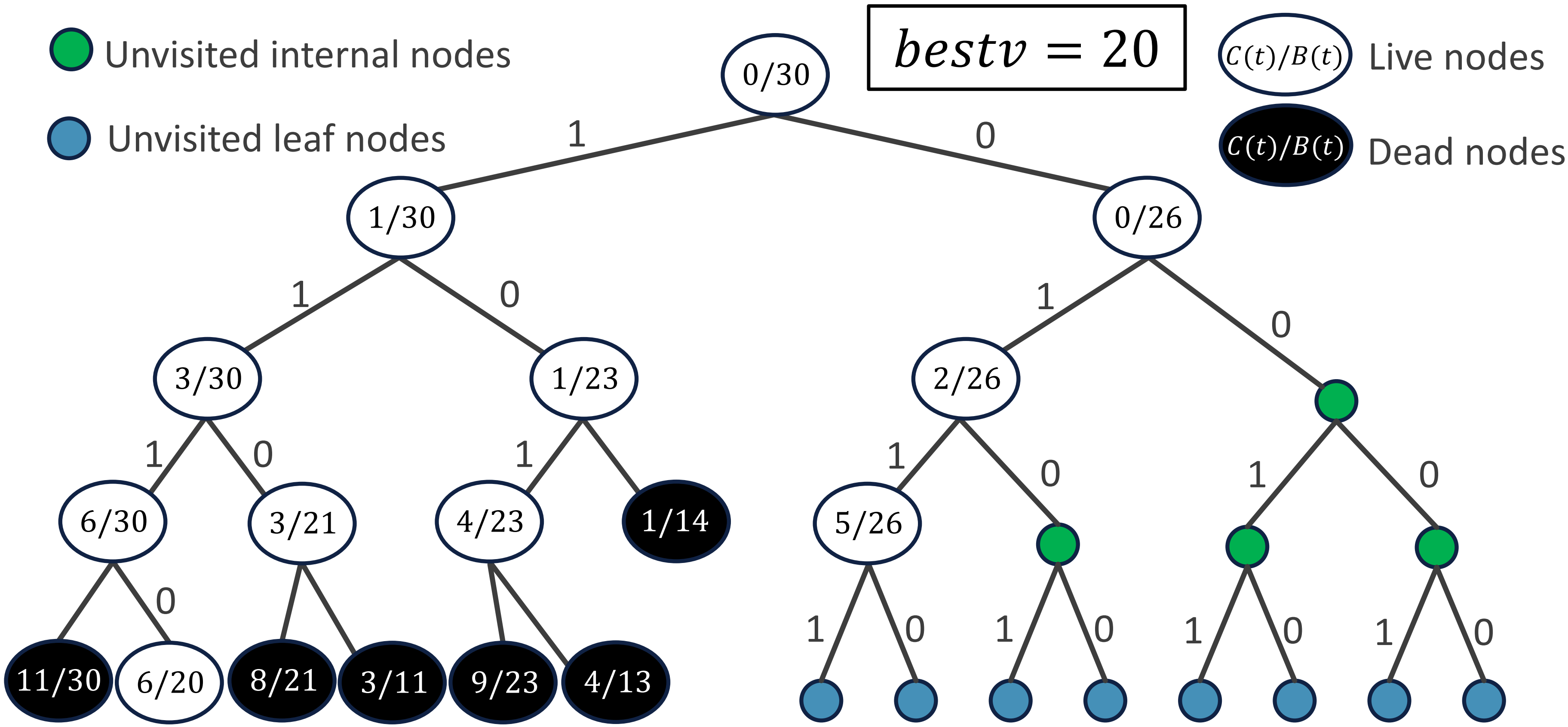
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



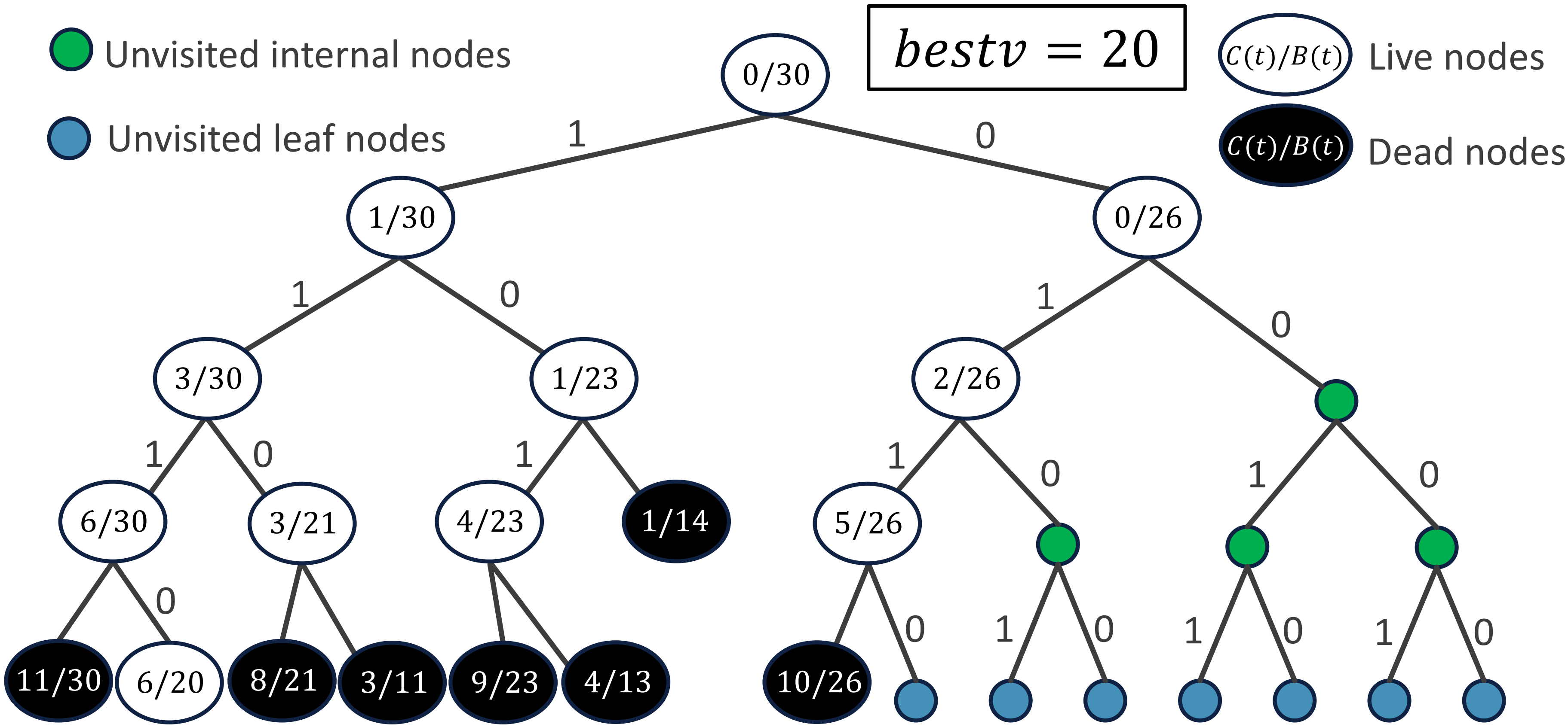
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



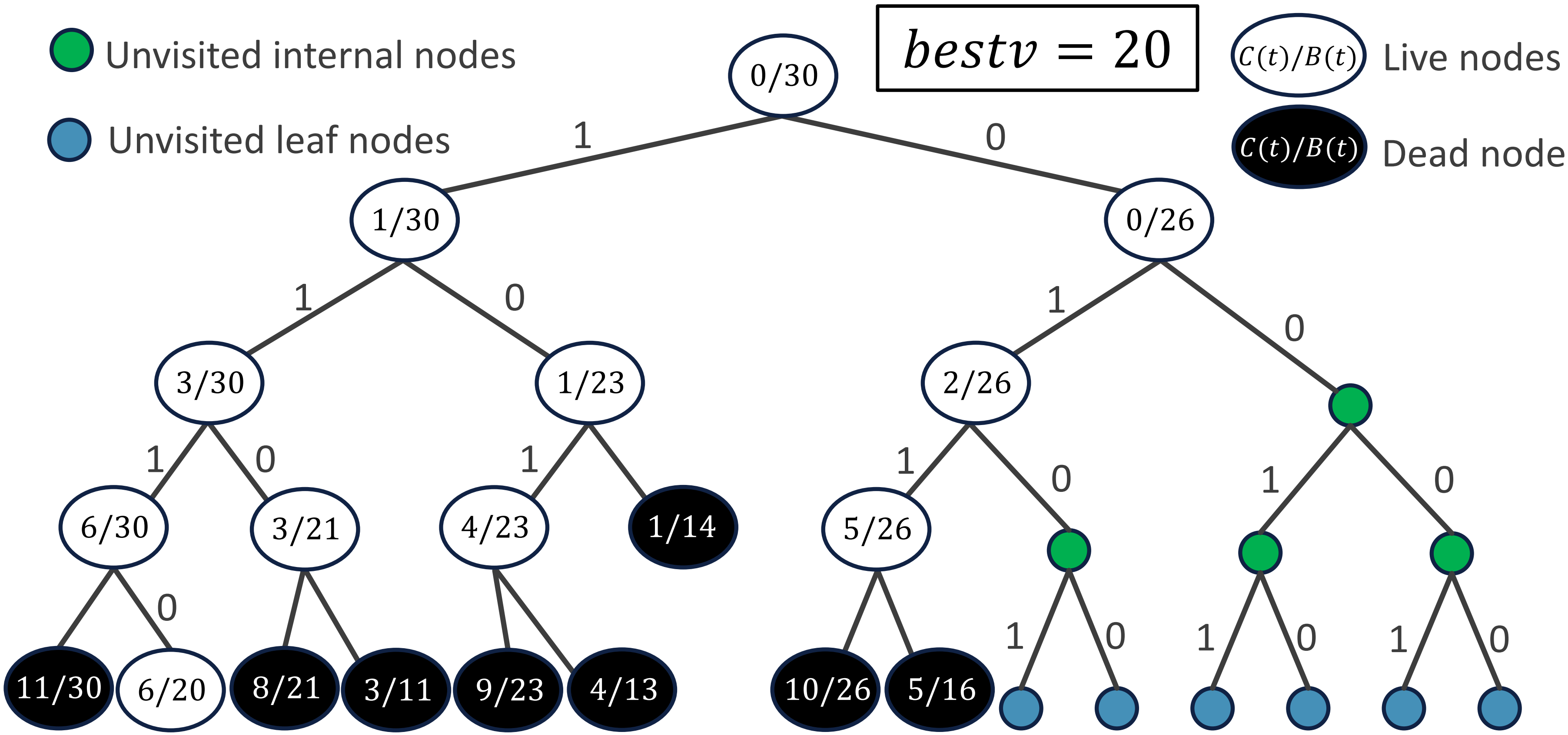
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



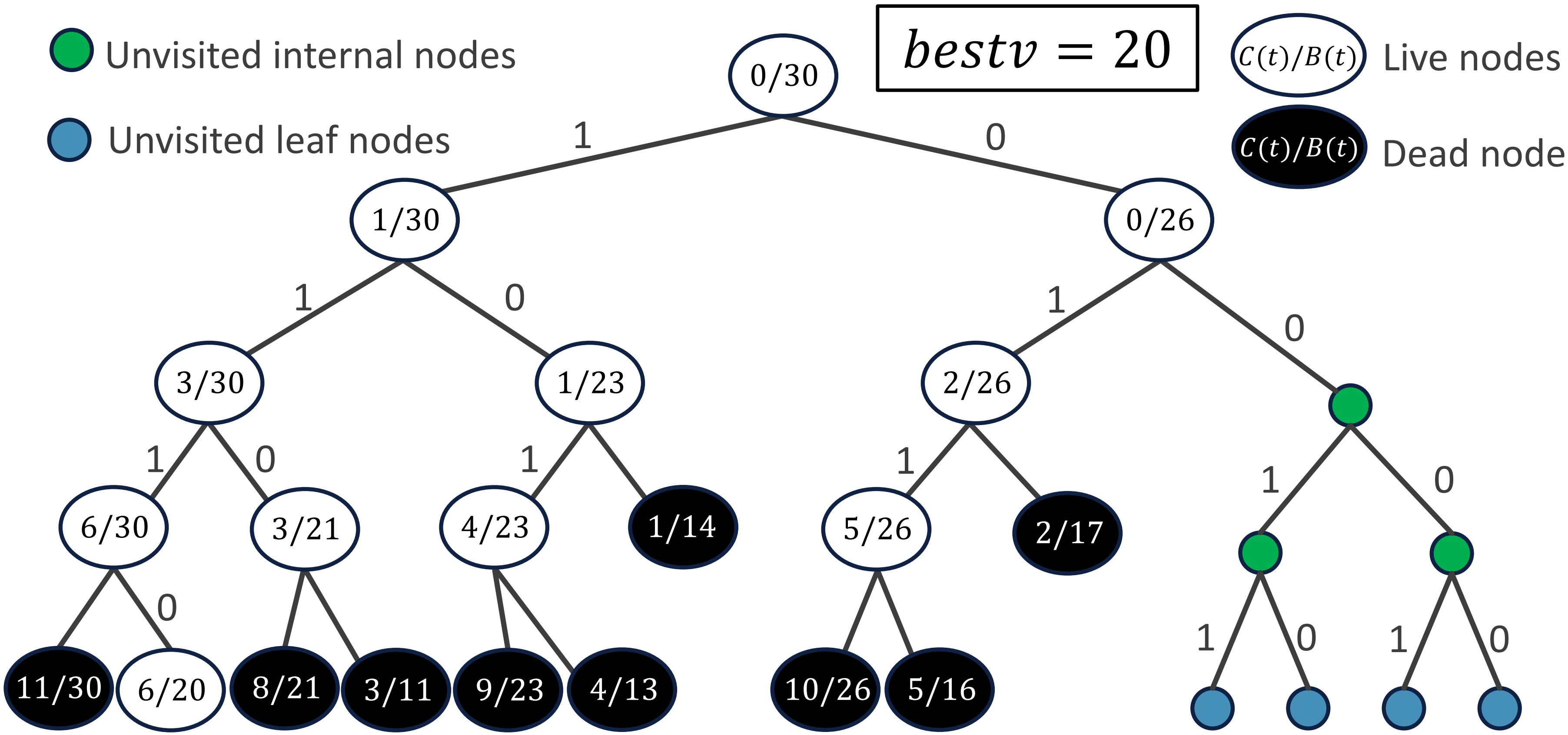
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



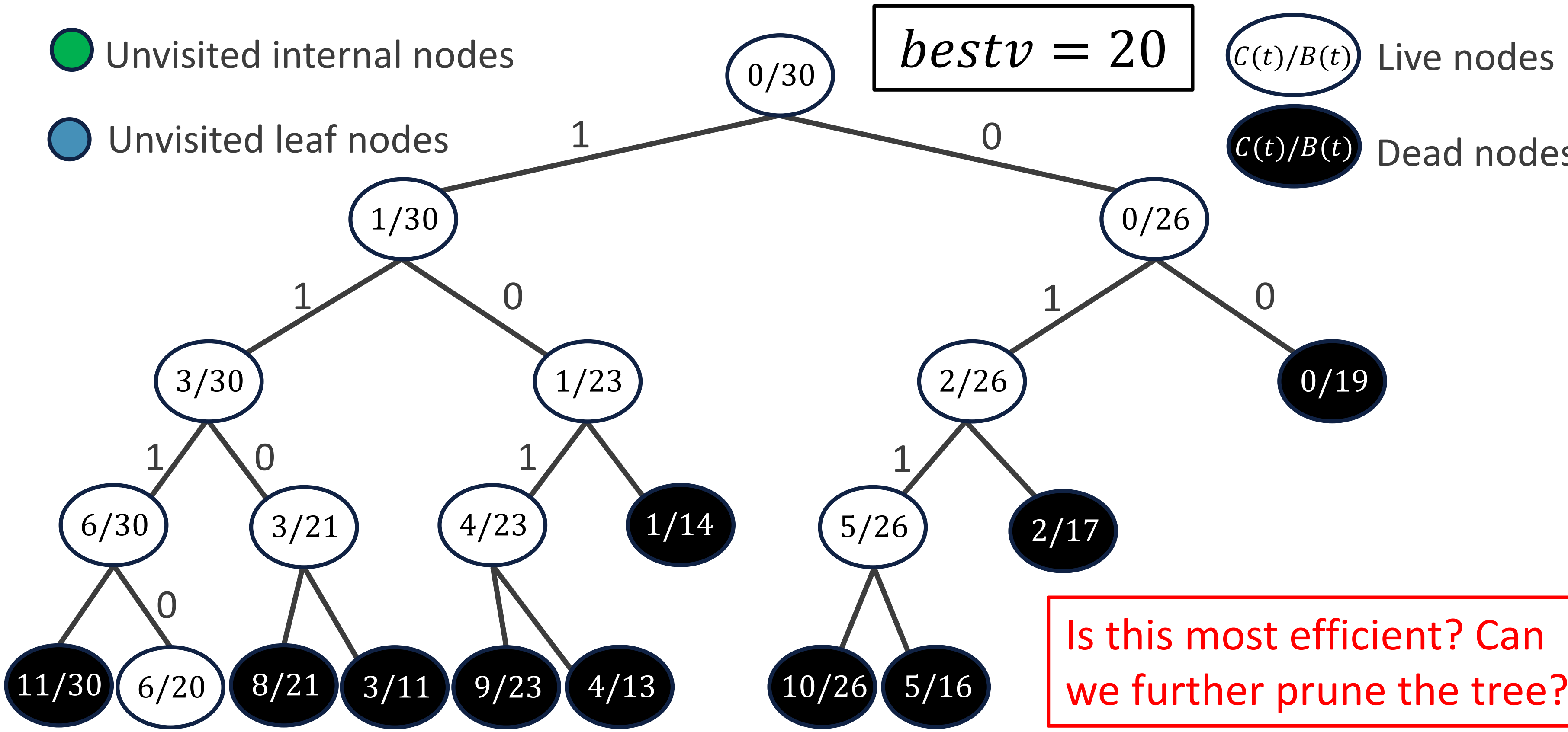
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

Example



Is this most efficient? Can we further prune the tree?

Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

0/1 Knapsack Problem

- Let's look back at the bounding function

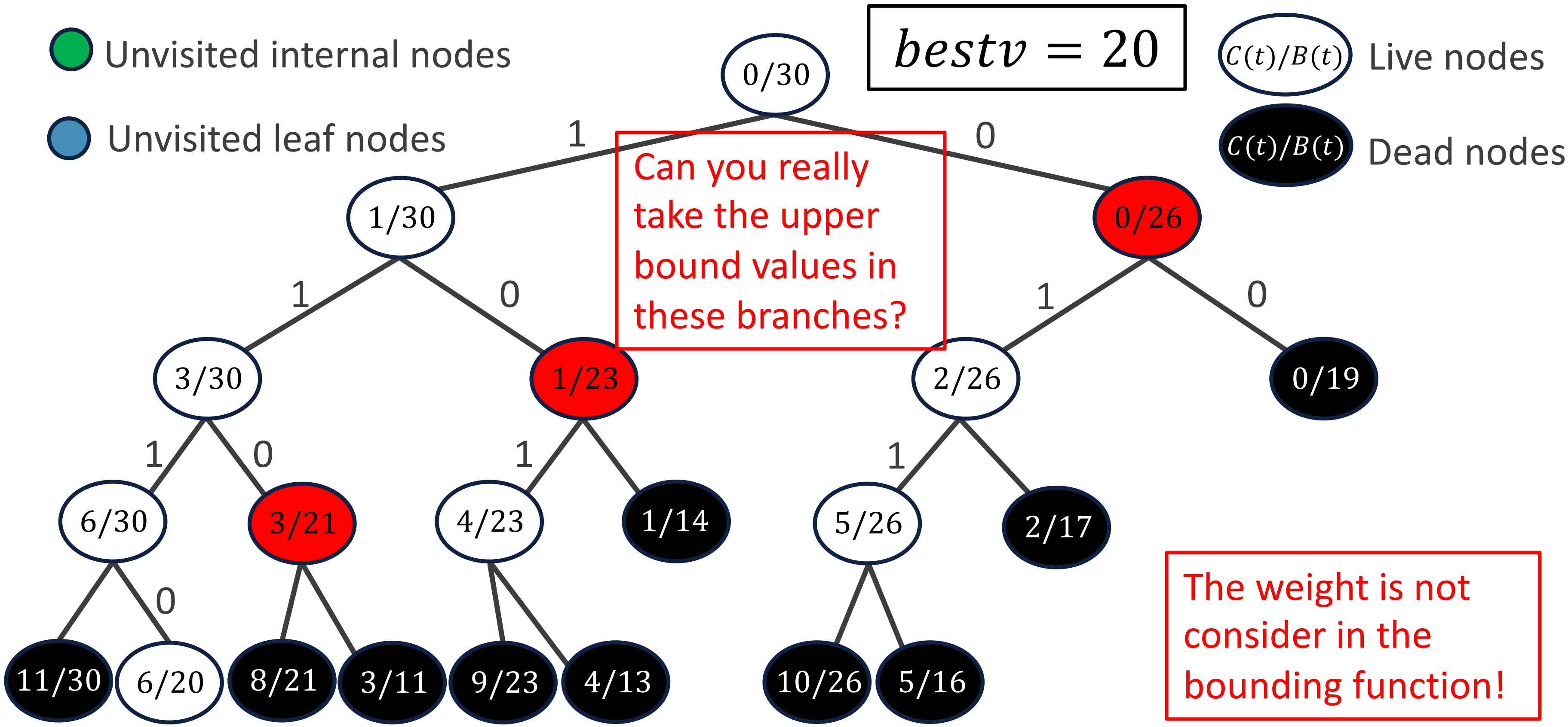
$$B(i) = cv(i) + r(i) \qquad cv(i) = \sum_{j=1}^{i} v_j x_j \qquad r(i) = \sum_{j=i+1}^{n} v_j$$

with the condition $B(i) \leq bestv$.

- What can we do if we want to prune more branches?

Make the bound tighter by decreasing the value of $B(i)$ (actually $r(i)$), because $C(i)$ is fixed at level i).

Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$

0/1 Knapsack Problem

- Now, we consider the weight limit in the bounding function.
- Given the remaining capacity $W - cw(i)$, what is the maximum value can we get?
- We can use the following greedy strategy:
  - Take the most valuable remaining items until we can't take any more.
  - Take a fraction of the next item until fully loaded.
- It does not mean we can really take fraction of item. It is just the upper bound of the remaining value.

0/1 Knapsack Problem

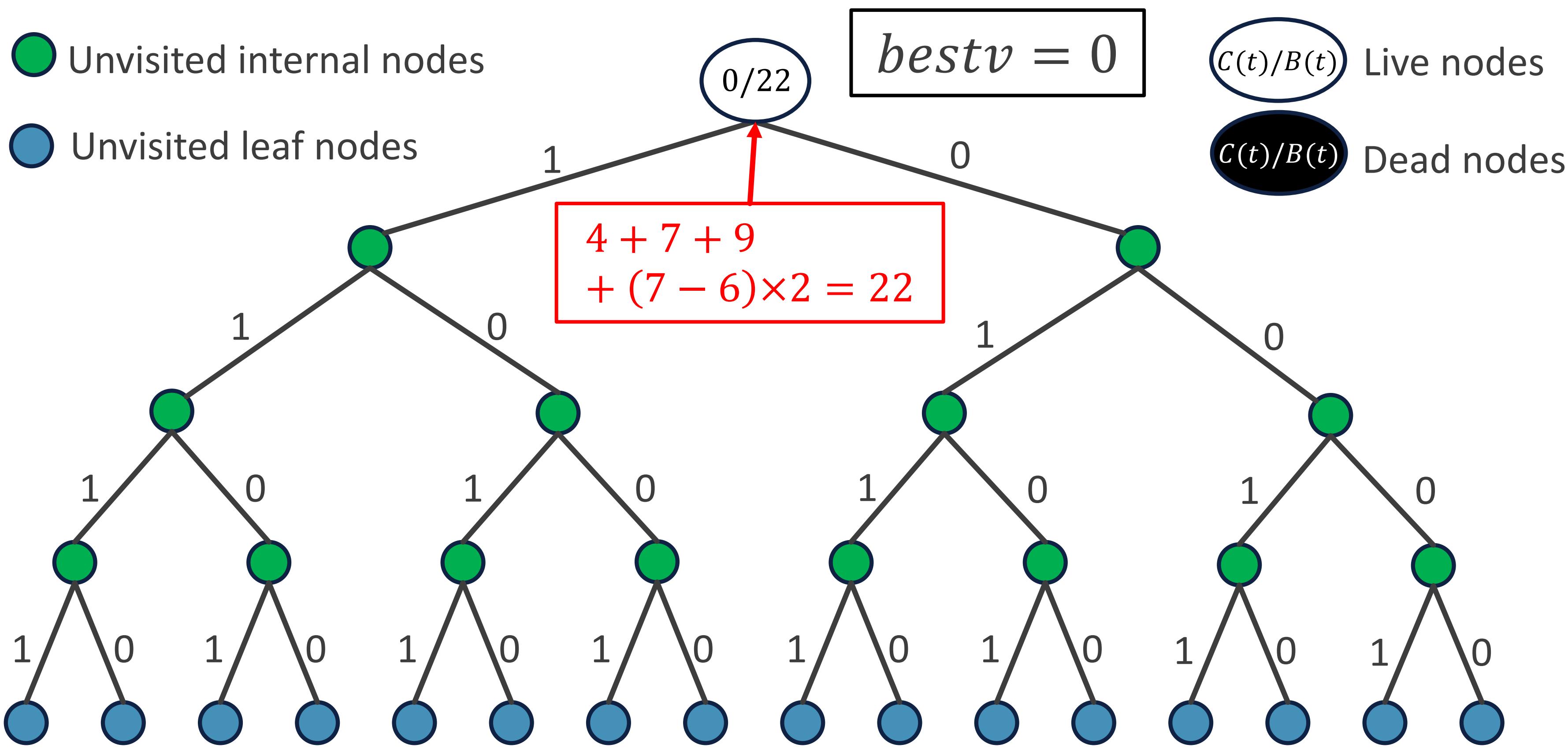
- First, sort the objects in decreasing order of value/weight ahead of time, namely

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$$

- Now, we are at level i, which means we have made decision for the first i items.
- We continue to put from item $i+1$ until item k. When put item k in, the load exceeds W.
- Then we take a fraction of item k for the remaining capacity.

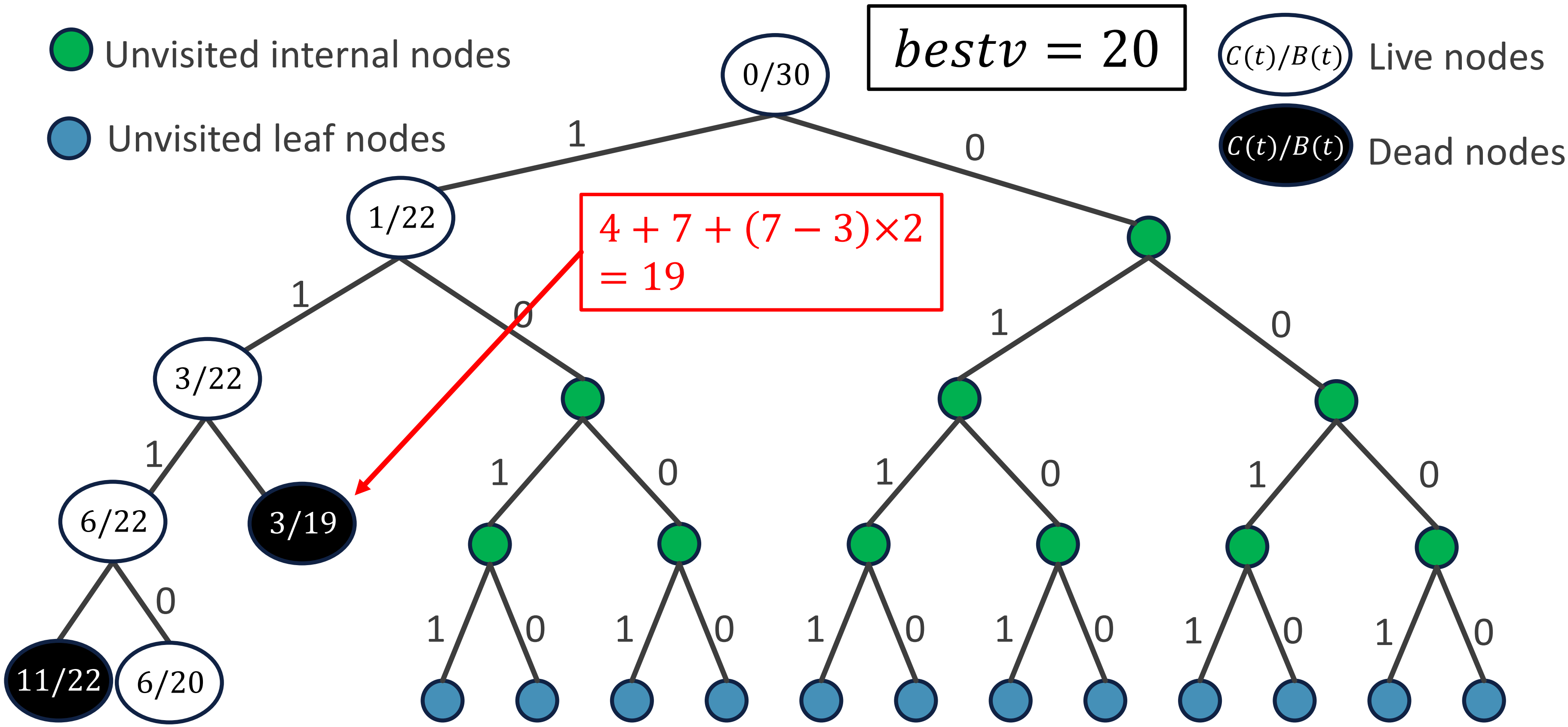
$$r(i) = \underbrace{\sum_{j=i+1}^{k-1} v_j}_{\text{Total value from item } i+1 \text{ to } k-1} + \underbrace{(W - cw(i) - \sum_{j=i+1}^{k-1} w_j\ )}_{\text{Capacity available for item } k}\underbrace{(\frac{v_k}{w_k})}_{\text{Value per unit weight for item } k}$$

Example



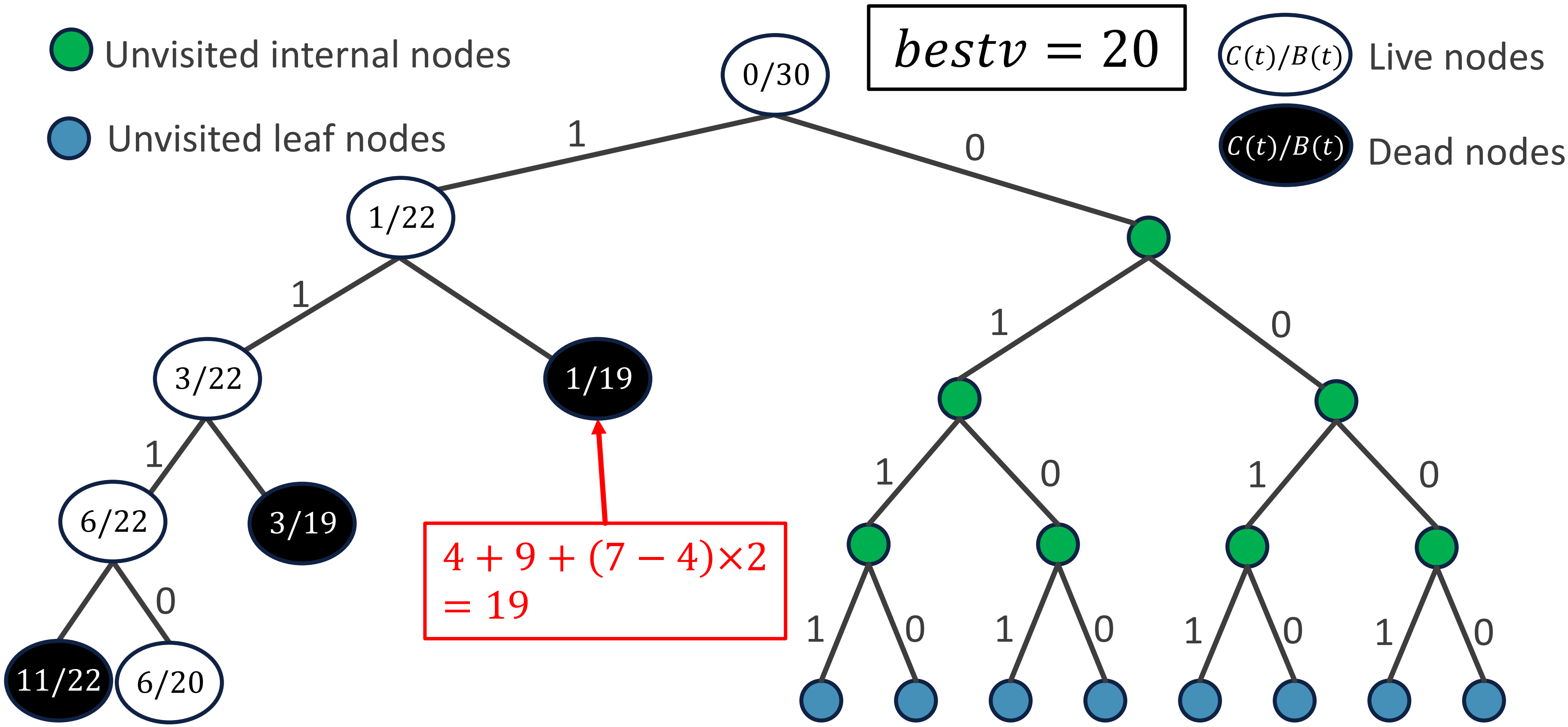
Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$, $v/w = [4, 3.5, 3, 2]$

Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$, $v/w = [4, 3.5, 3, 2]$

Example



Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$, $v/w = [4, 3.5, 3, 2]$

Example

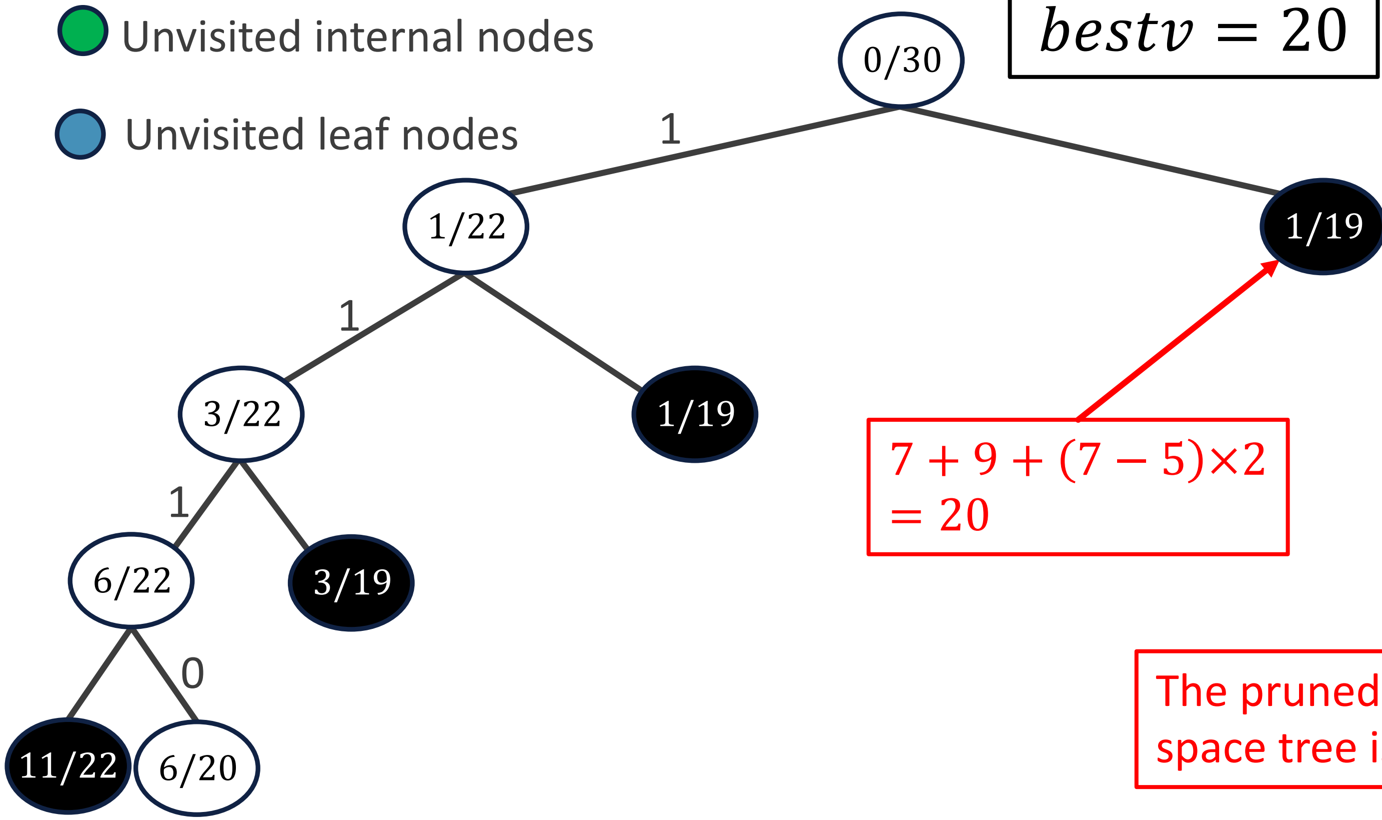
Unvisited internal nodes

Unvisited leaf nodes

$bestv = 20$

$C(t)/B(t)$ Live nodes

$C(t)/B(t)$ Dead nodes



The pruned solution space tree is much better!

Backtracking for $n = 4$, $v = [4,7,9,10]$, $w = [1,2,3,5]$, $W = 7$, $v/w = [4, 3.5, 3, 2]$

Pseudocode

BacktrackKnapsack(i)

if $i > n$ **then**
    if $cv > bestv$ **then**
        $bestv \leftarrow cv$
        for $j \leftarrow 1$ **to** n **do**
            $bestx[j] \leftarrow x[j]$
else
    if $C(i) \leq W$ **then** $x[i] \leftarrow 1$
        $cw \leftarrow cw + w[i]$; $cv \leftarrow cv + v[i]$;
        BacktrackKnapsack($i + 1$)
        $cw \leftarrow cw - w[i]$; $cv \leftarrow cv - v[i]$;
    if $B(i) > bestv$ **then** $x[i] \leftarrow 0$
        BacktrackKnapsack($i + 1$)

We don't record the remaining value here and leave it in $B(i)$.

Nothing special compared with container loading problem. Just separate cw and cv.

Pseudocode

$r(i)$

$rw \leftarrow W - cw$ Remaining capacity

$b \leftarrow cv$ Total value

while $i + 1 \leq n$ and $w[i+1] \leq rw$ **do** Loop until we can't take the whole item $i + 1$

  $rw \leftarrow rw - w[i+1]$

  $b \leftarrow b + v[i+1]$

  $i \leftarrow i + 1$ Take a fraction of item $i + 1$

if $i + 1 \leq n$ **then** $b \leftarrow b + v[i+1]/w[i+1] \times rw$

return b

Classroom Exercise

- Draw the pruned solution space tree of 0/1 knapsack problem for the following problem instance:

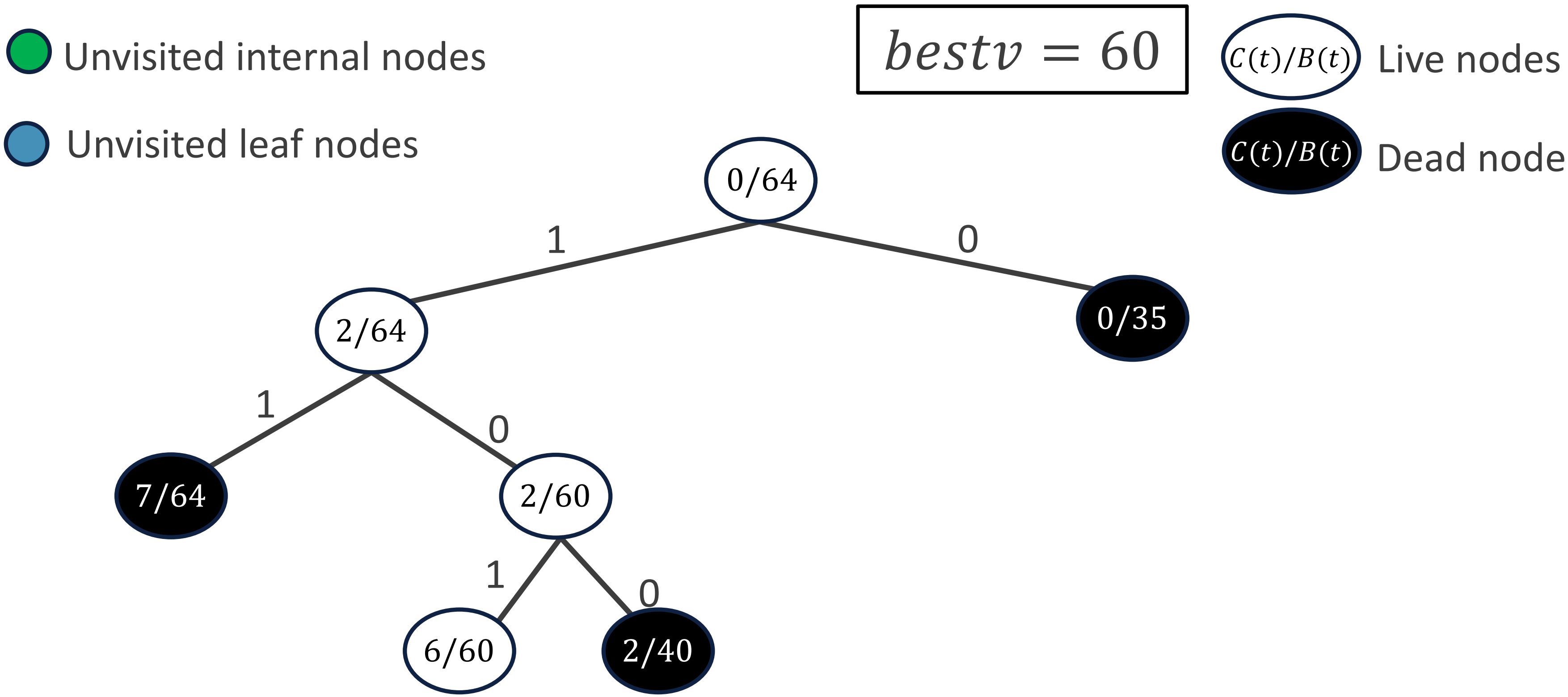
$$n = 3, v = [4,3,1], w = [2,5,5], W = 6$$

Classroom Exercise

- First, rank the item by their value per unit weight:

$n = 3, v = [40,30,20], w = [2,5,4], W = 6, v/w = [20,6,5]$

Classroom Exercise

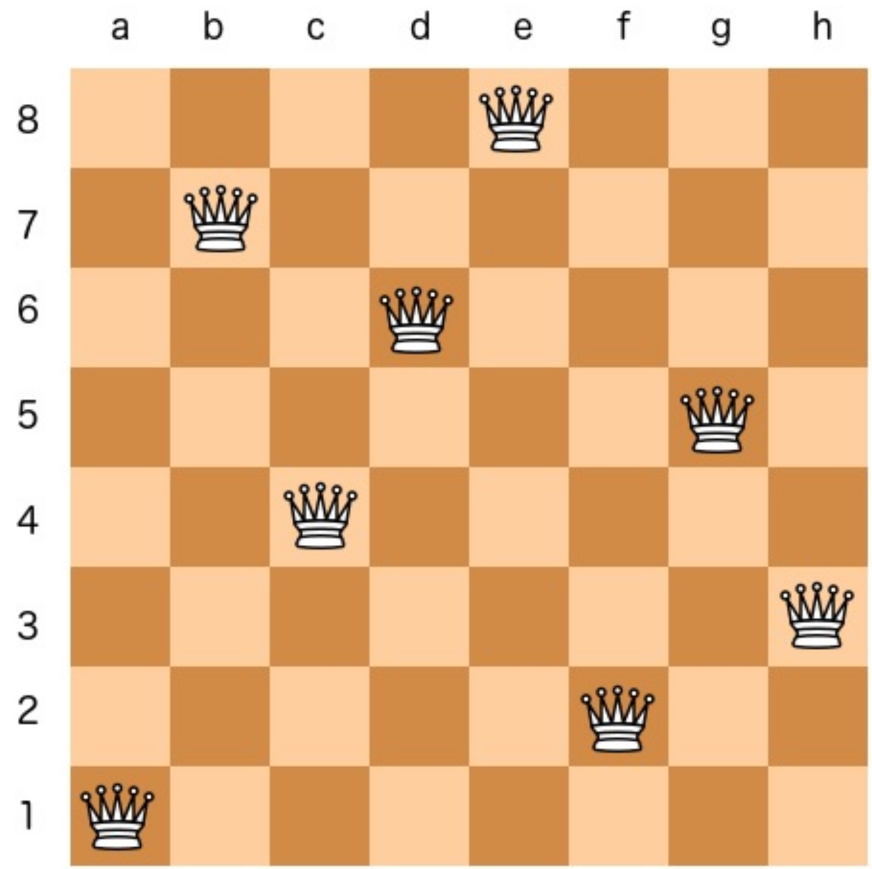


Backtracking for $n = 3, v = [40,30,20], w = [2,5,4], W = 6, v/w = [20,6,5]$

n QUEEN PROBLEM

n Queen Problem

- The goal of n queen problem (n皇后问题) is to position n queens on an $n \times n$ chessboard so that no two queens threaten each other.
  - No two queens may be in the same row, column, or diagonal.



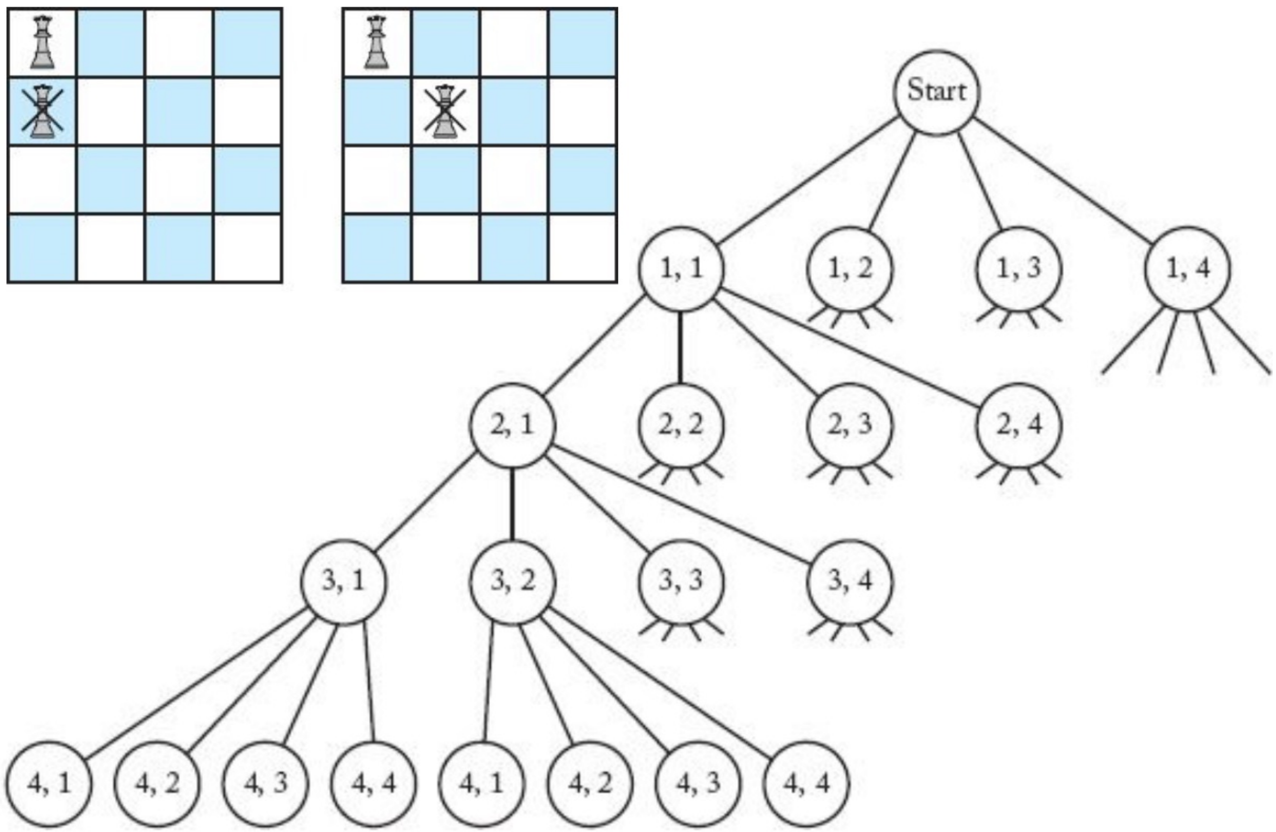
厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: https://medium.com/swlh/how-many-solutions-does-the-n-queens-problem-have-e8da5d45a34c

n Queen Problem

- What is the size of solution space for the ith queen?
- $n^2 - i + 1$? It is too large. We can limit it by considering the constraint.
  - Because two queens can't be put in the same row, we directly put each queen in different row.
  - Now, the solution space for the ith queen is n.
  - Thus, the constraint function only needs to check if two queens are in the same column or diagonal.

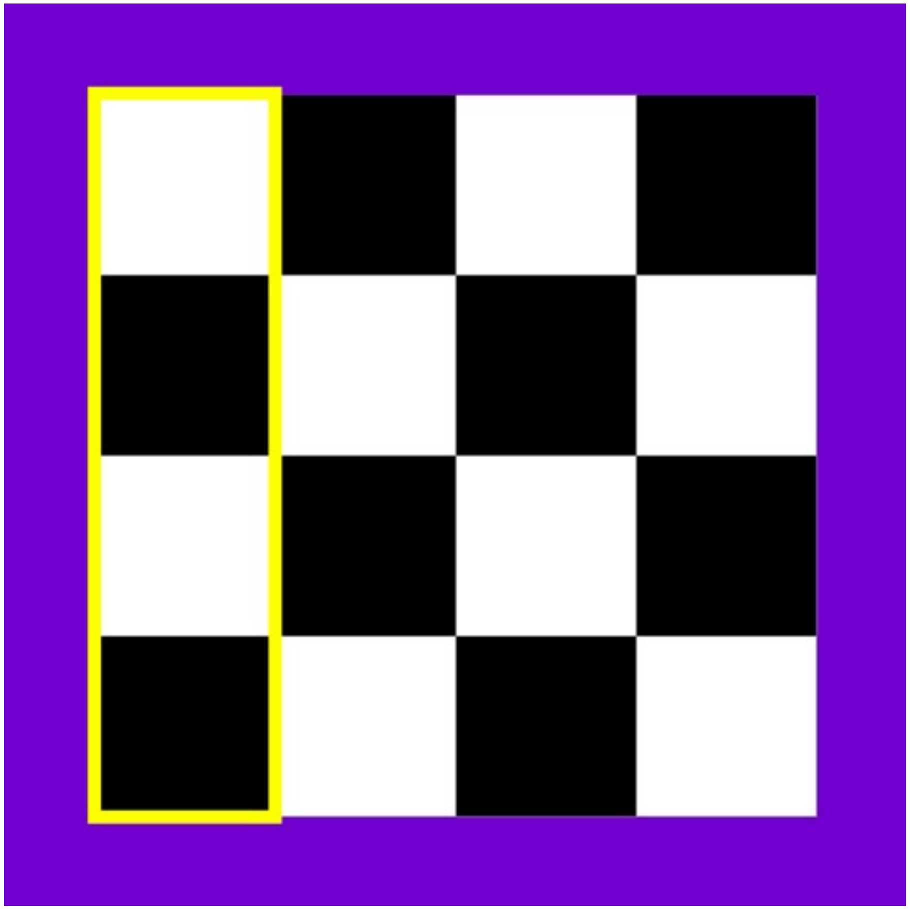
n Queen Problem



The solution space tree for $n = 4$

Image source: Figure 5.2-5.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

n Queen Problem



What is the constraint function?

Image source: https://meetwithbudhi.wordpress.com/2019/09/16/n-queens-puzzle/

n Queen Problem

- The constraint function checks if the new added queen is in the same column, or along the same diagonal.
- Now, we know that the ith queen is in the ith row. Let x_i be the column of the ith queen.
  - If the kth and jth queen are in the same column:

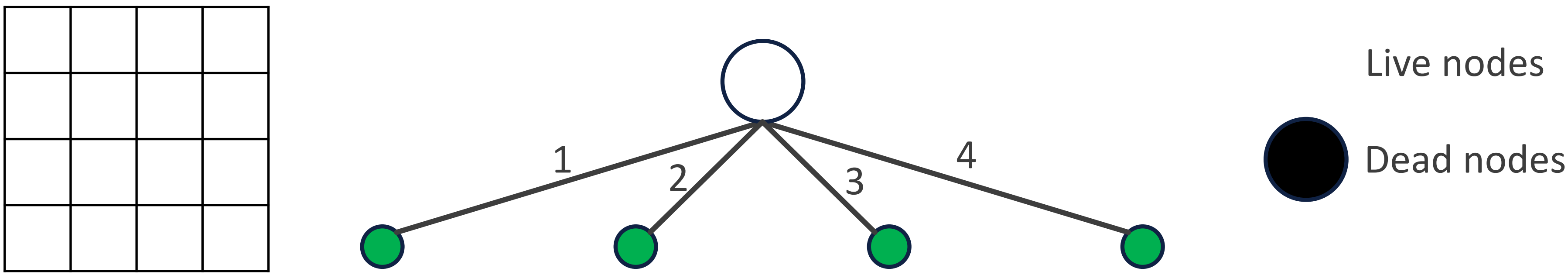
$$x_k = x_j$$

  - If the kth and jth queen are along the same diagonal:

$$x_k - x_j = k - j \quad \text{or} \quad x_k - x_j = j - k$$

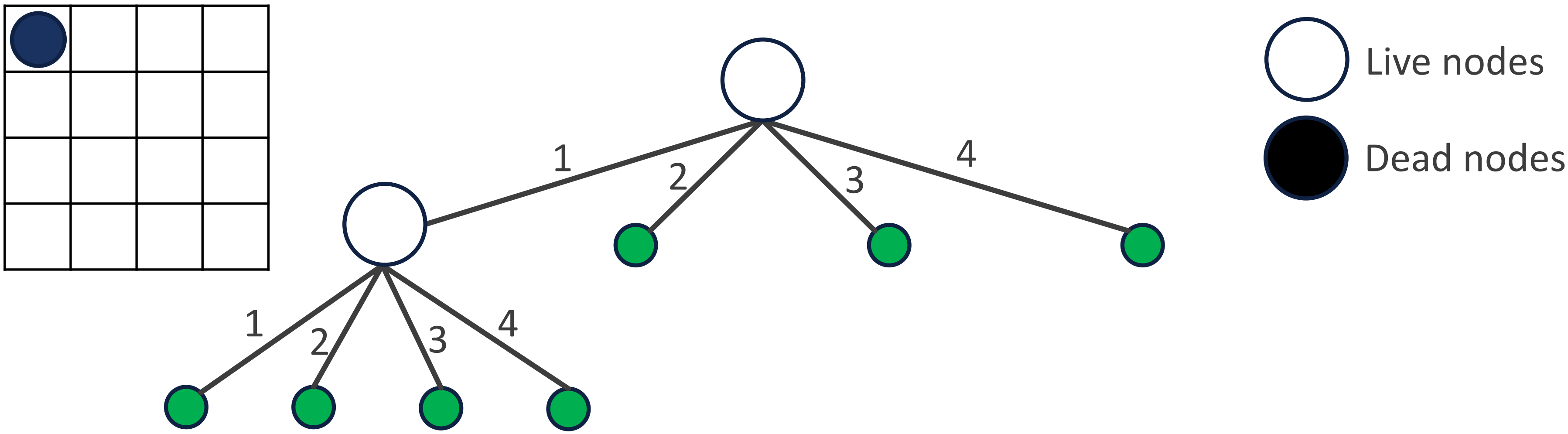
Namely: $|x_k - x_j| = |k - j|$.

Example



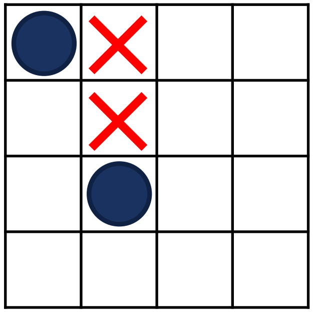
Backtracking for $n = 4$

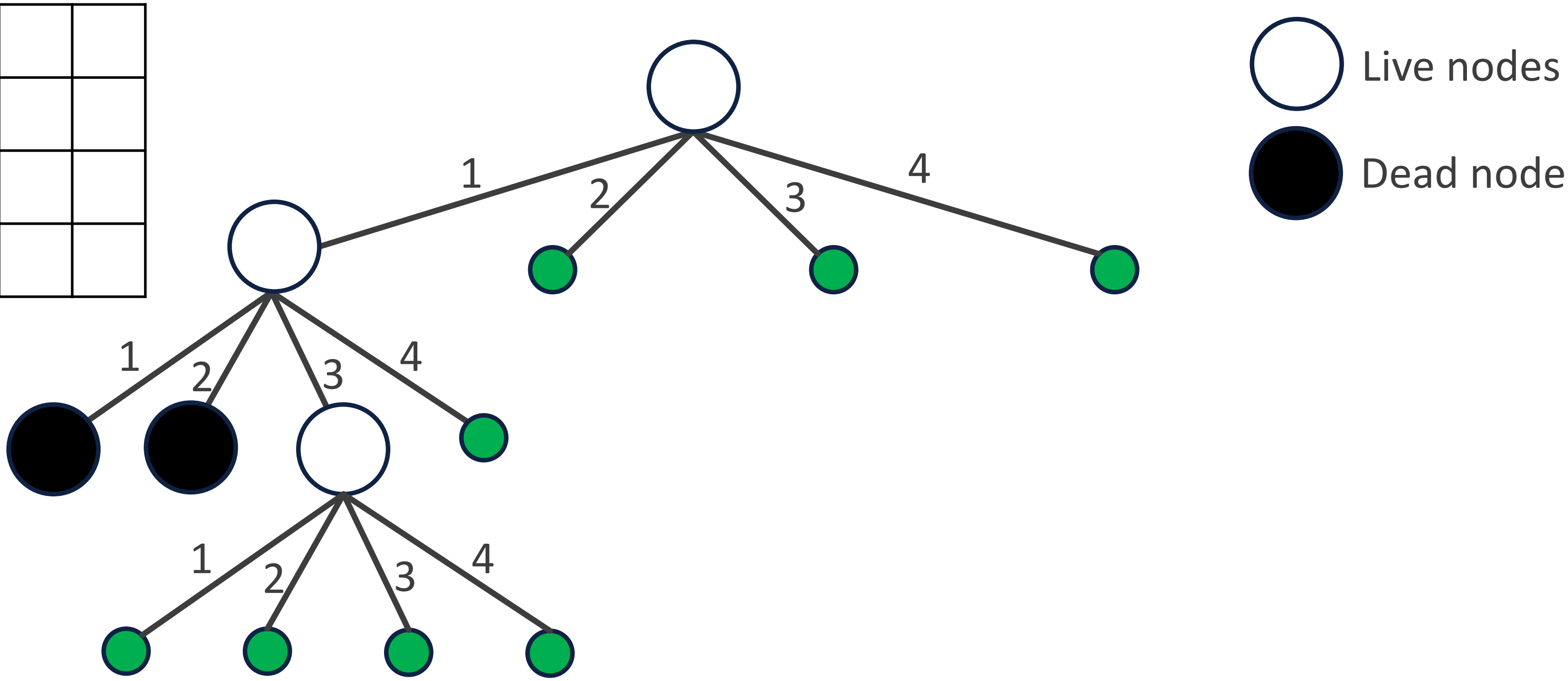
Example



Backtracking for $n = 4$

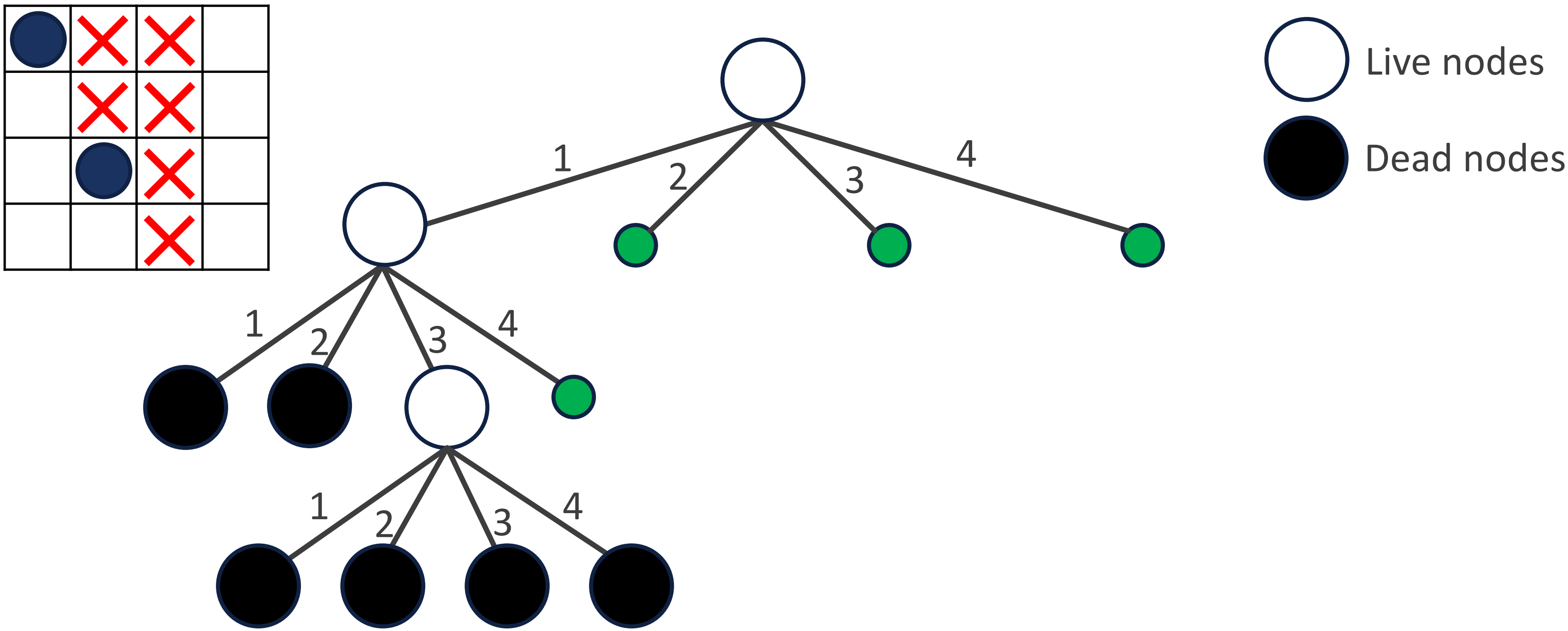
Example





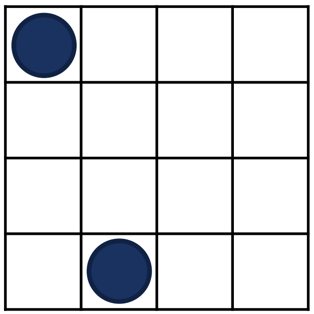
Backtracking for $n = 4$

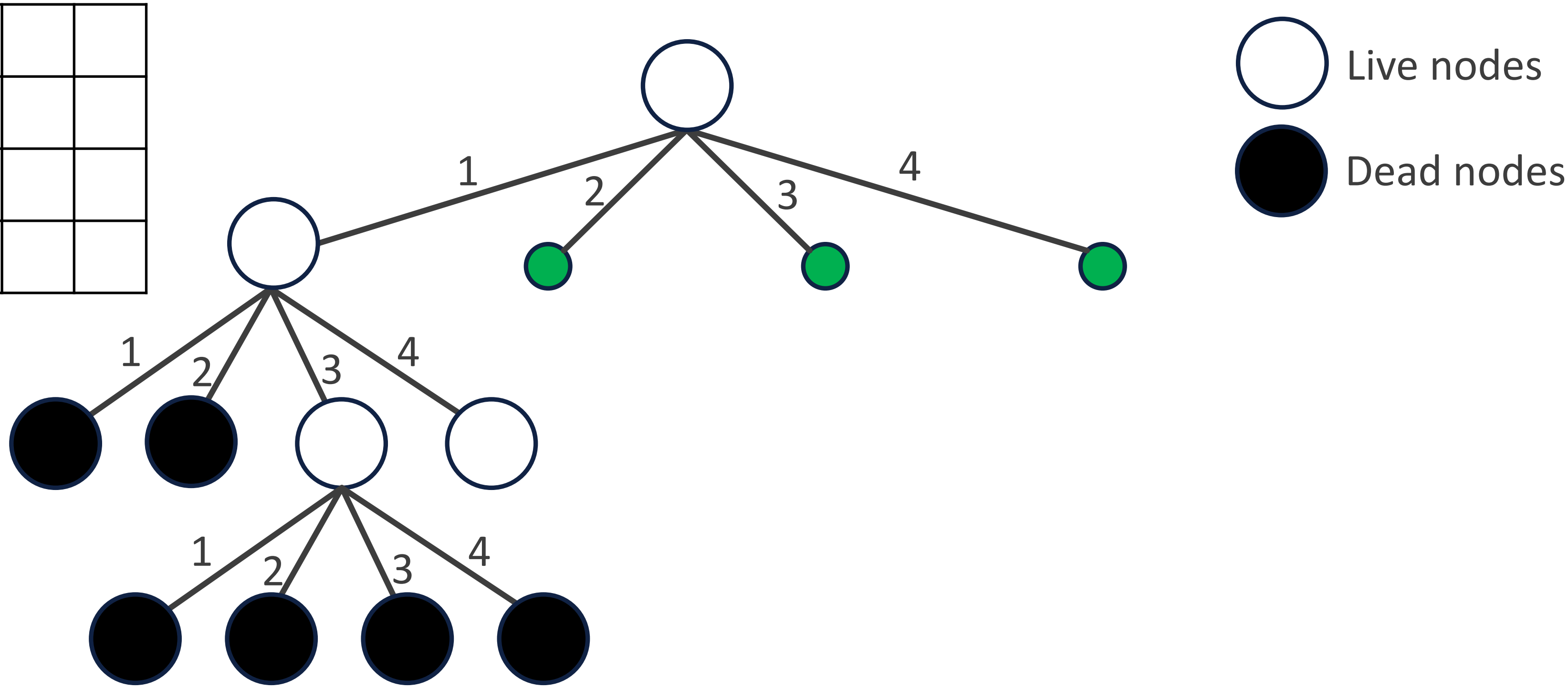
Example



Backtracking for $n = 4$

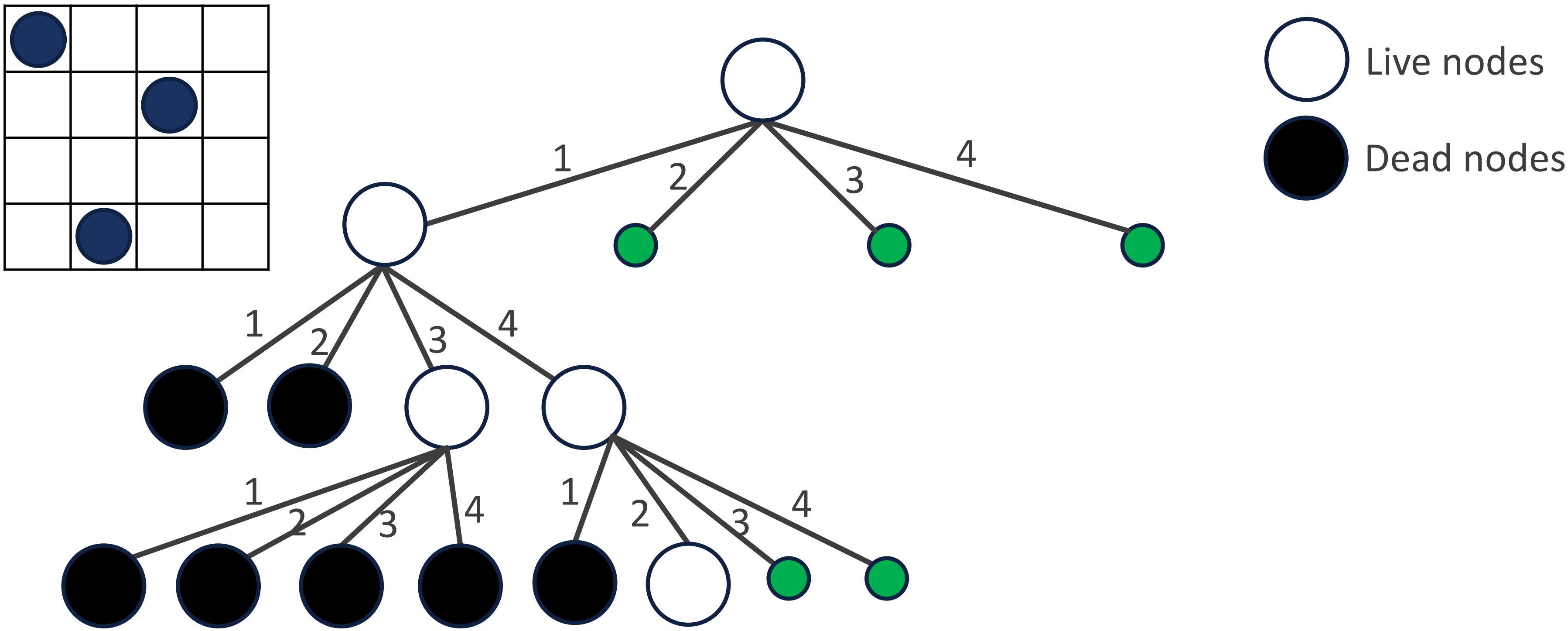
Example





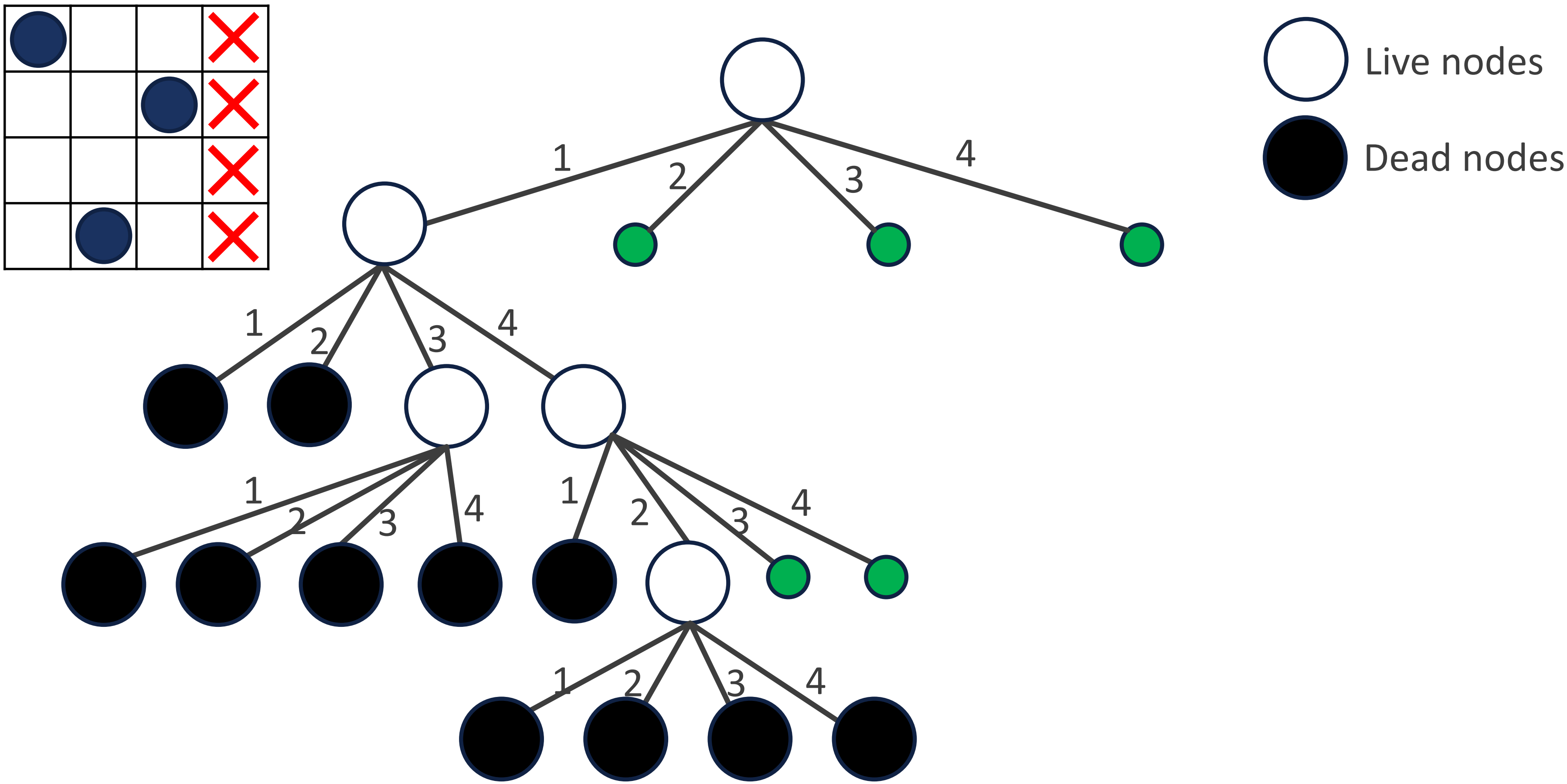
Backtracking for $n = 4$

Example



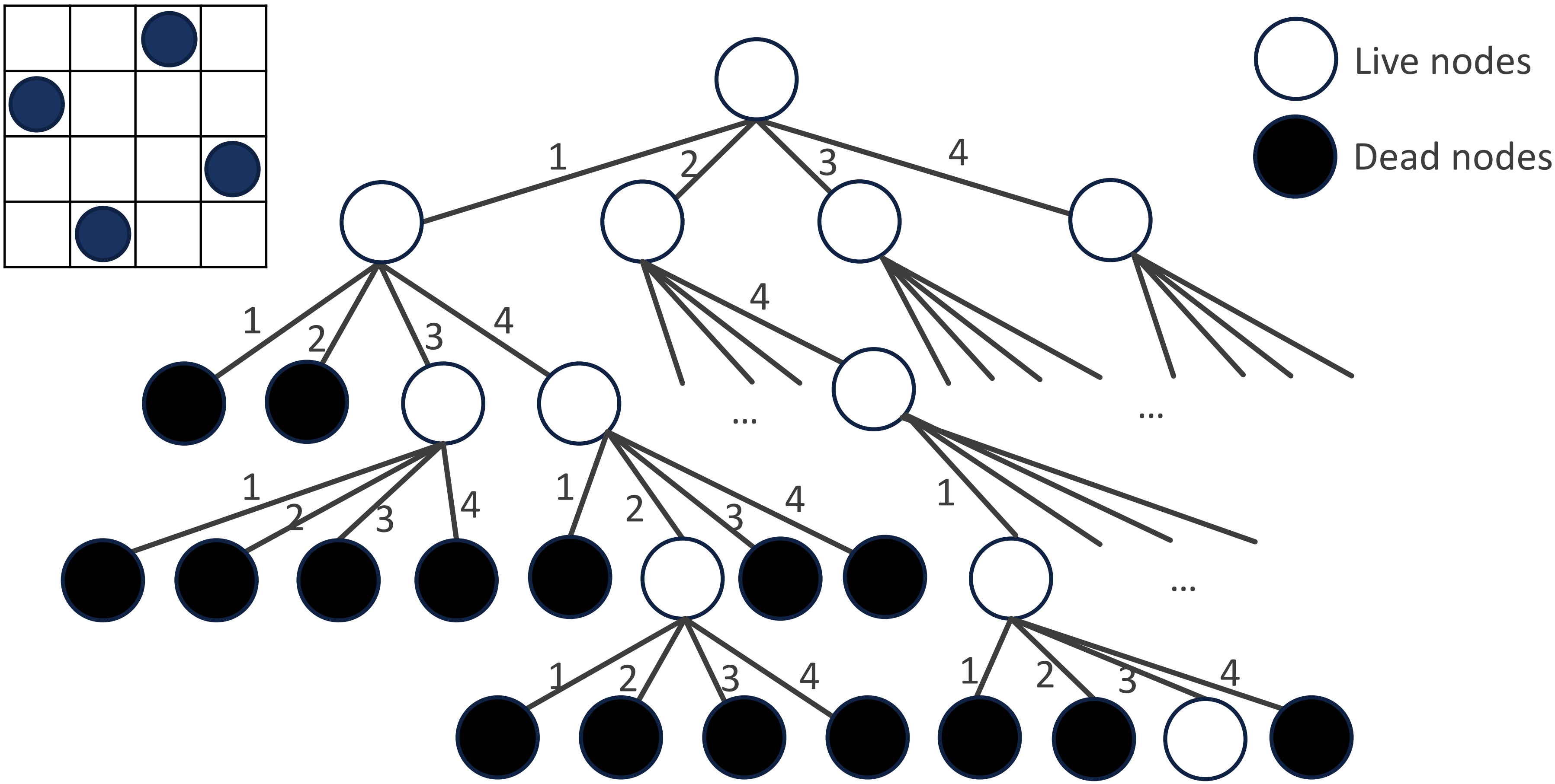
Backtracking for $n = 4$

Example



Backtracking for $n = 4$

Example



Backtracking for $n = 4$

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系
Computer Science Department of Xiamen University

Pseudocode

BacktrackNqueens()

$x[1] \leftarrow 0$
$k \leftarrow 1$
while $k > 0$ **do**
    while $x[k] \leq n-1$ **do**
        $x[k] \leftarrow x[k] + 1$
        if Place(k) = True **then**
            if $k = n$ **then** $SolNum \leftarrow SolNum + 1$
            else
                $k \leftarrow k + 1$
                $x[k] \leftarrow 0$
    $k \leftarrow k - 1$

Start from 0, increment in the loop, so the condition only checks $\leq n-1$

Number of valid solutions

Place(k)

for $j \leftarrow 1$ **to** $k-1$ **do**
    if $|k-j| = |x[k] - x[j]|$ or $x[j] = x[k]$ **then**
        return False
return True

No recursion is used here

Classroom Exercise

Write the pseudocode of the recursive version of n queen problem.

Classroom Exercise

```
RecursiveBacktrackNqueens(k)
    if Place(k) = True then
        if k = n then  SolNum ← SolNum + 1
        else
            for j ← 1 to n do
                x[k + 1] ← j
                RecursiveBacktrackNqueens(k + 1)
```

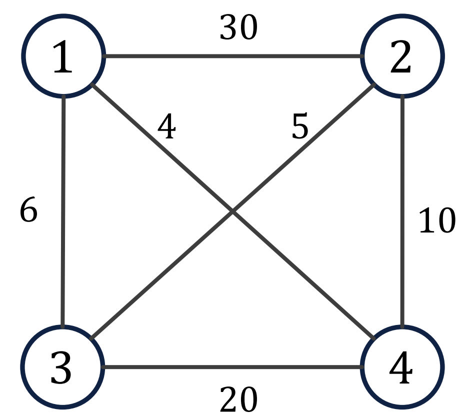
Start from 0

```
Place(k)
    for j ← 1 to k − 1 do
        if |k − j| = |x[k] − x[j]| or x[j] = x[k] then
            return False
    return True
```

TRAVELING SALESPERSON PROBLEM

Traveling Salesperson Problem

- Given an n vertex network (undirected or directed), traveling salesperson problem (旅行商问题, TSP) is to find a cycle of minimum cost that includes all n vertices.
  - Hamiltonian cycle with minimum cost.
- Any cycle that includes all n vertices of a network is called a tour. In TSP, we are to find a least-cost tour. For example:
  - Tour (1,2,4,3,1) costs 66.
  - Tour (1,4,3,2,1) costs 59.
  - Tour (1,3,2,4,1) costs 25, optimal.



Traveling Salesperson Problem

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: https://www.programmersought.com/article/32324255027/

Traveling Salesperson Problem

- Since a tour is a cycle that includes all vertices, we may pick any vertex as the start (and hence the end).
  - Usually we use vertex 1 as the start and end vertex.
- Each tour is then described by the vertex sequence:

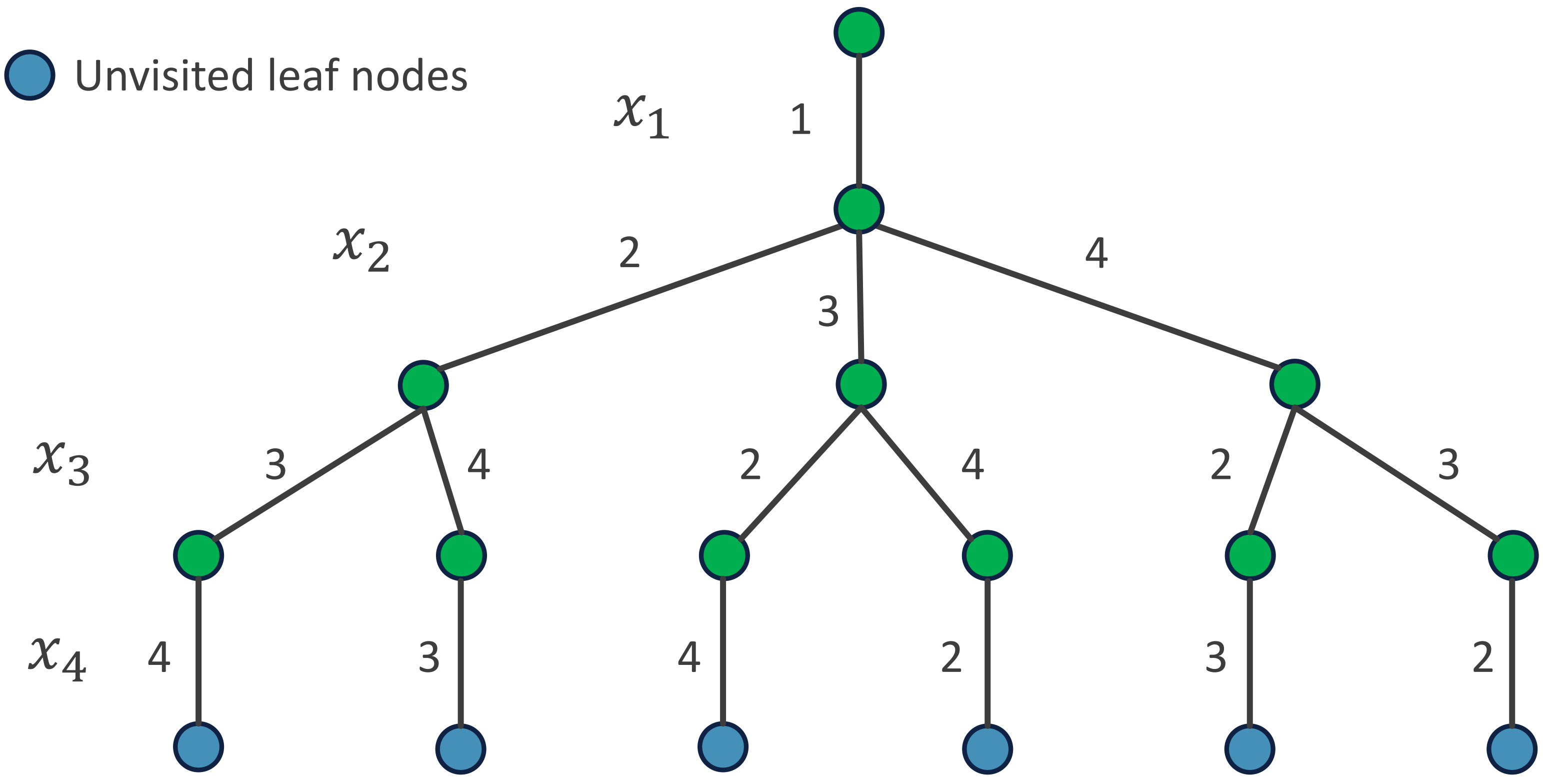
$$(1, x_2, \dots, x_n, 1)$$

  where $x_2, \dots, x_n$ is a permutation of $(2, 3, \dots, n)$.
- The possible tours may be described by a permutation tree in which each root-to-leaf path defines a tour.

Traveling Salesperson Problem

Unvisited internal nodes

Unvisited leaf nodes



Permutation tree for TSP when $n = 4$

Traveling Salesperson Problem

- $w[i, j]$ denotes the weight of vertex i and vertex j.
- $w[i, j] = \infty$ denotes no edge between vertex i and vertex j.
- $x[i]$ denotes the vertex to be searched.
- What are the constraint function and bounding function?

Traveling Salesperson Problem

- Constraint function $C(i)$ is to simply check if the next vertex is connected to the current vertex:

$$C(i) = w\big[x[i], x[j]\big]$$

Check if $C(i) \neq \infty$.

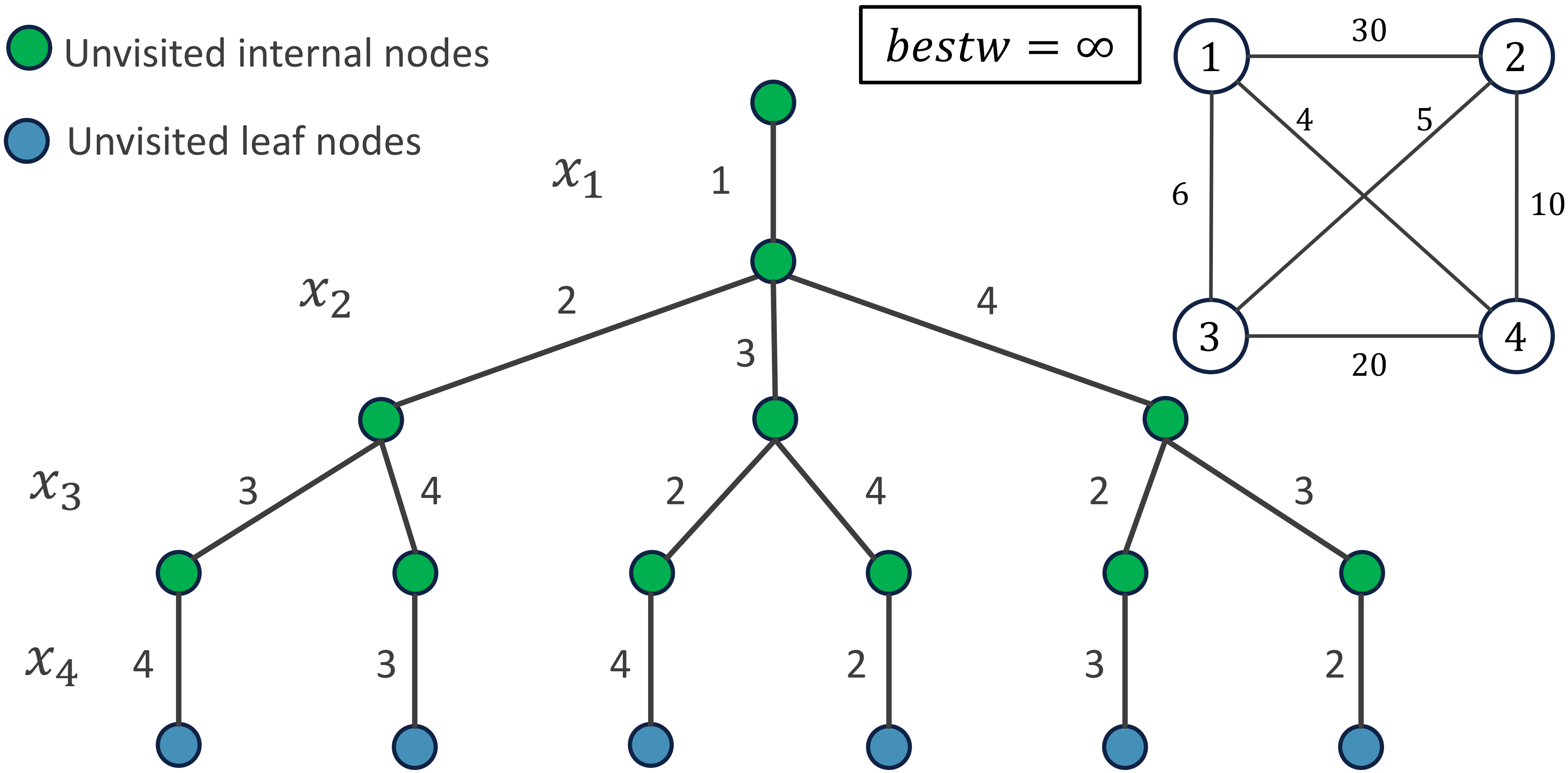
- Bounding function $B(i)$ is the total weight if we connect $x[i]$:

$$B(i) = cw(i-1) + w[x[i-1], x[i]]$$

$$cw(i) = \sum_{j=2}^{i} w[x[j-1], x[j]]$$

Check if $B(i) < bestw$.

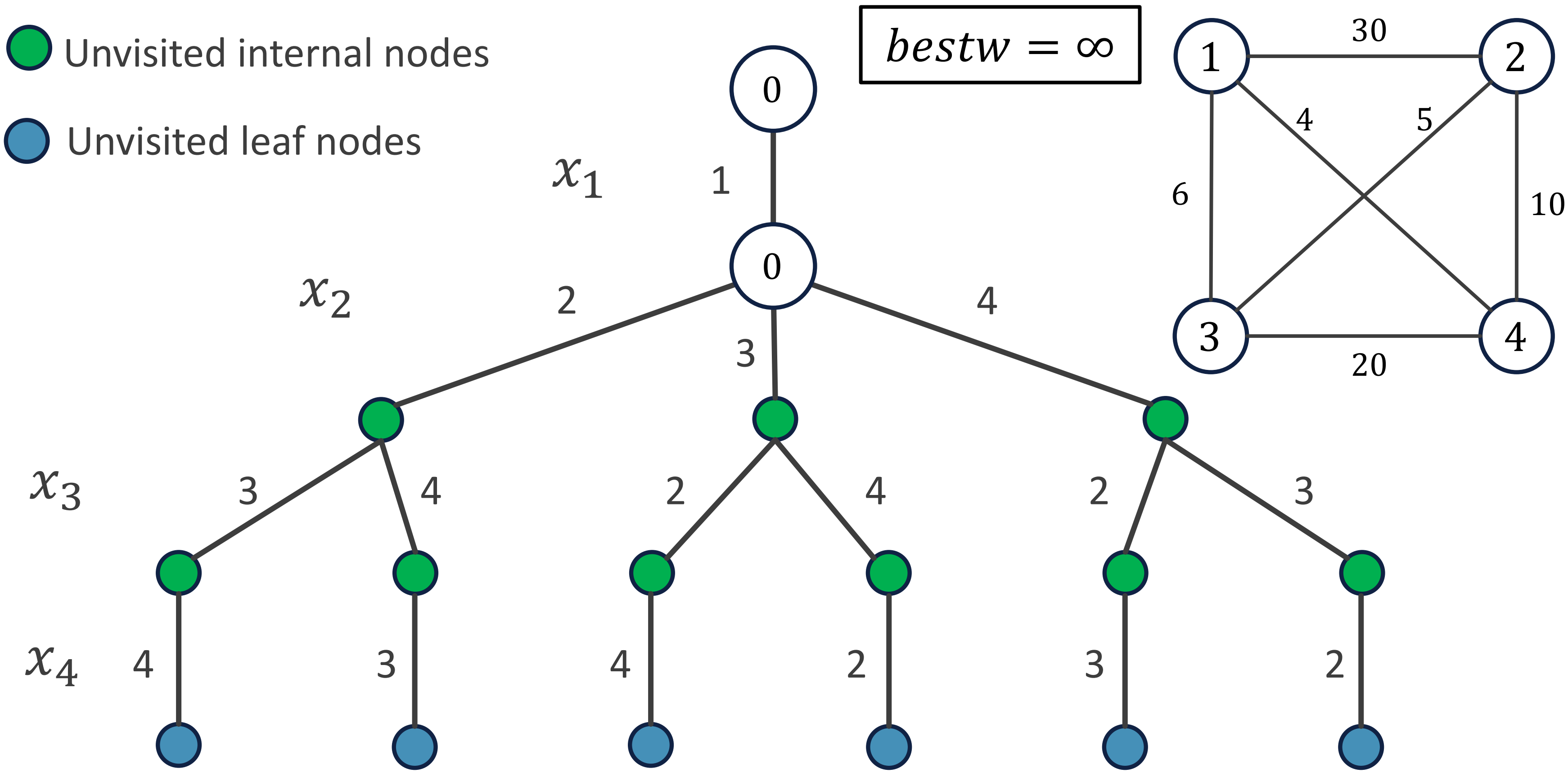
Example



This graph is complete graph. Therefore the constraint function is useless.

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Example



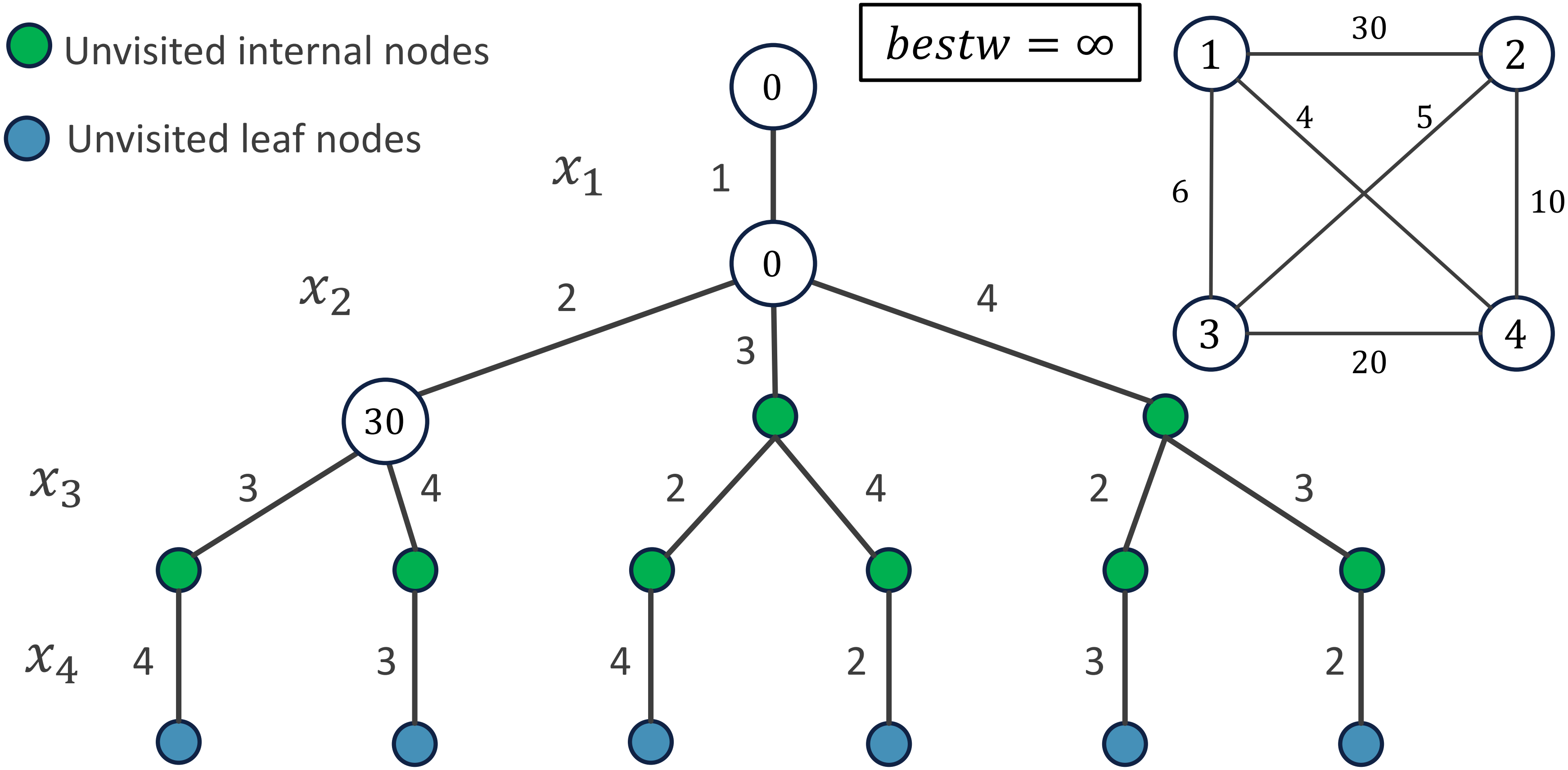
Unvisited internal nodes
Unvisited leaf nodes
$bestw = \infty$
0
1
0
x_1
x_2
x_3
x_4
2
3
4
3
4
2
4
2
3
4
3
4
2
3
2
1
2
3
4
30
4
5
6
10
20

Example

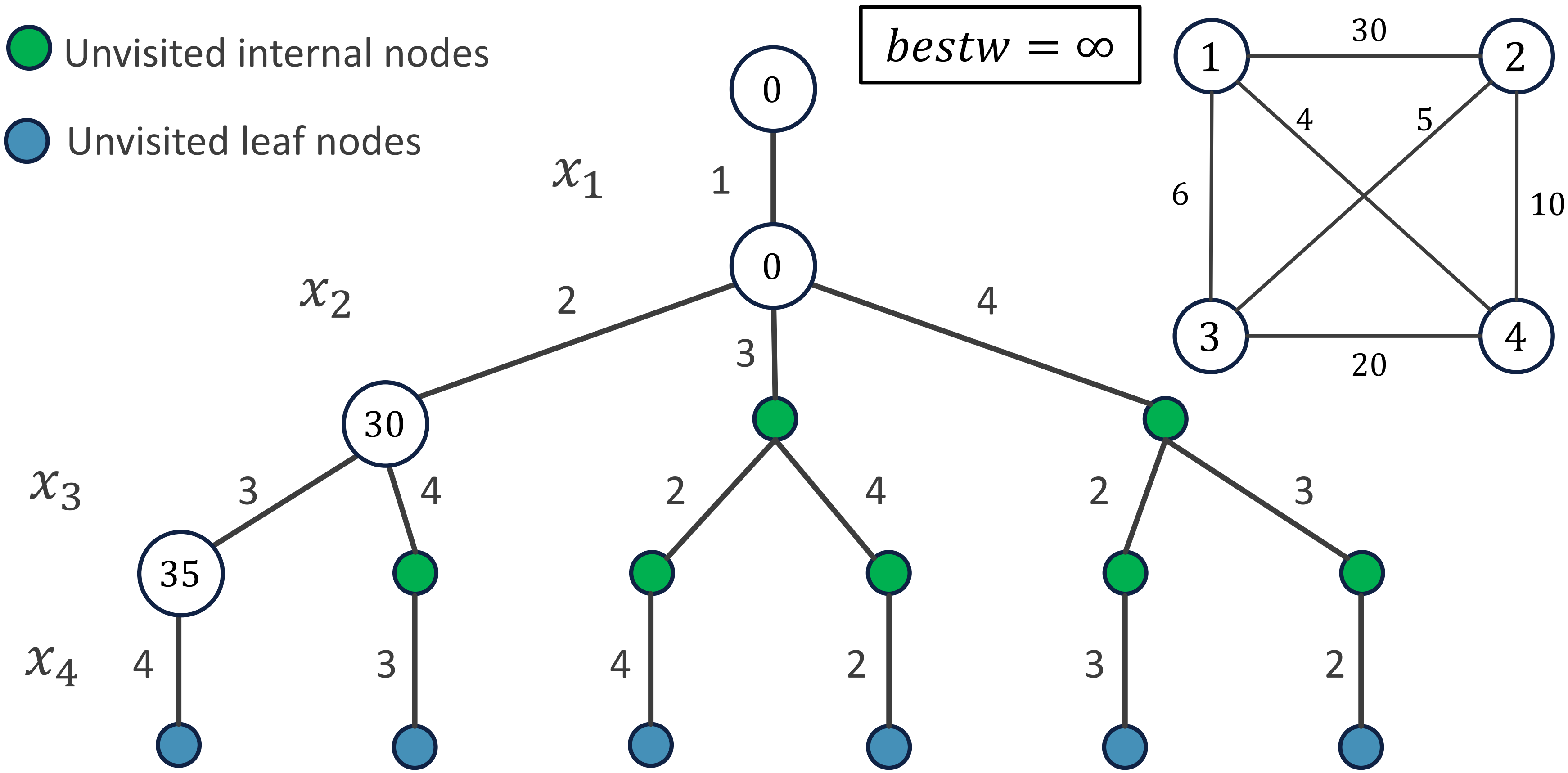
Unvisited internal nodes

Unvisited leaf nodes

$bestw = \infty$

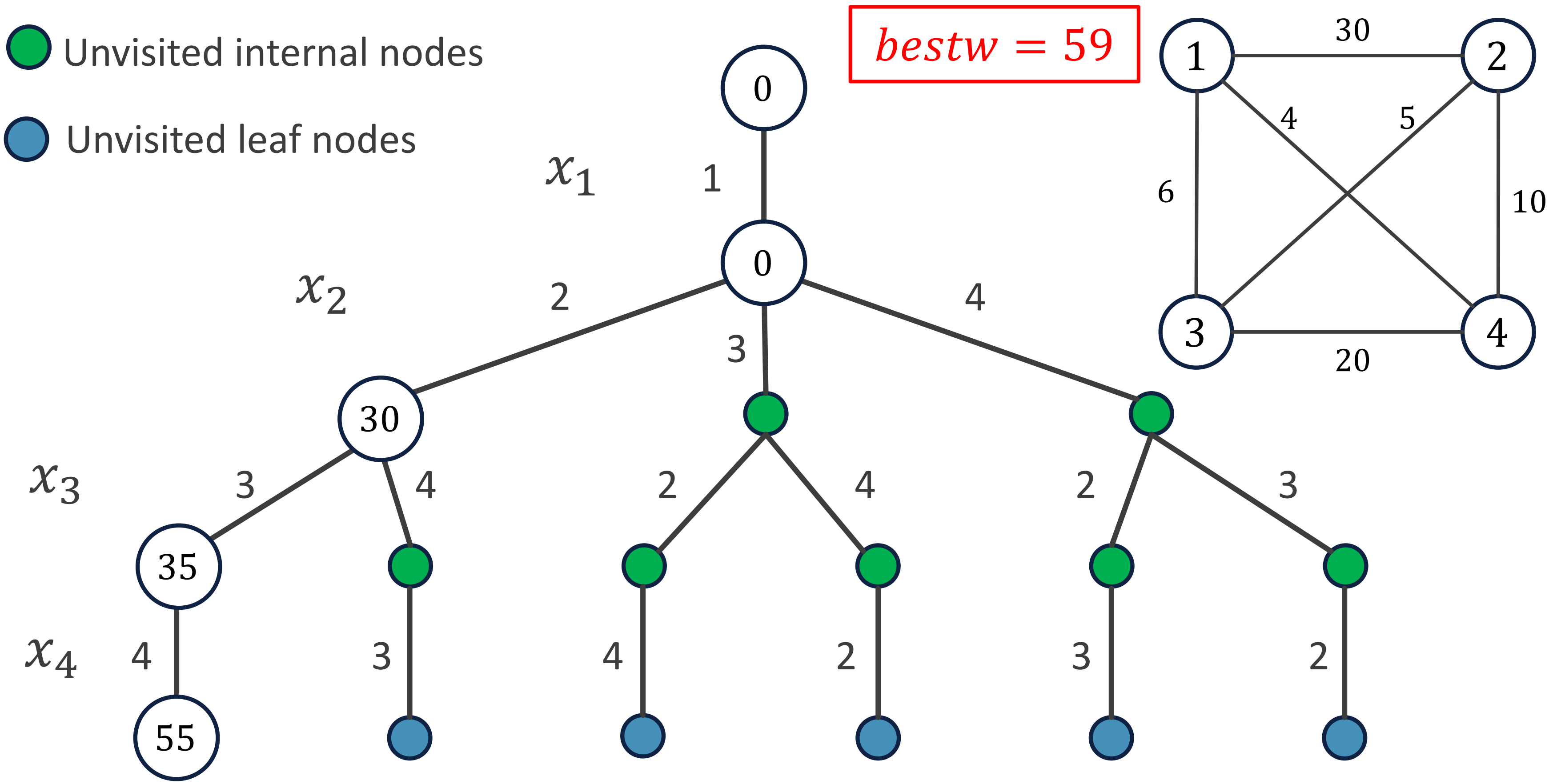


Example



Unvisited internal nodes
Unvisited leaf nodes
$bestw = \infty$
x_1
x_2
x_3
x_4
0
1
0
2
3
4
30
35
3
4
2
4
2
3
4
3
4
2
3
2
1
2
3
4
30
4
5
6
10
20

Example

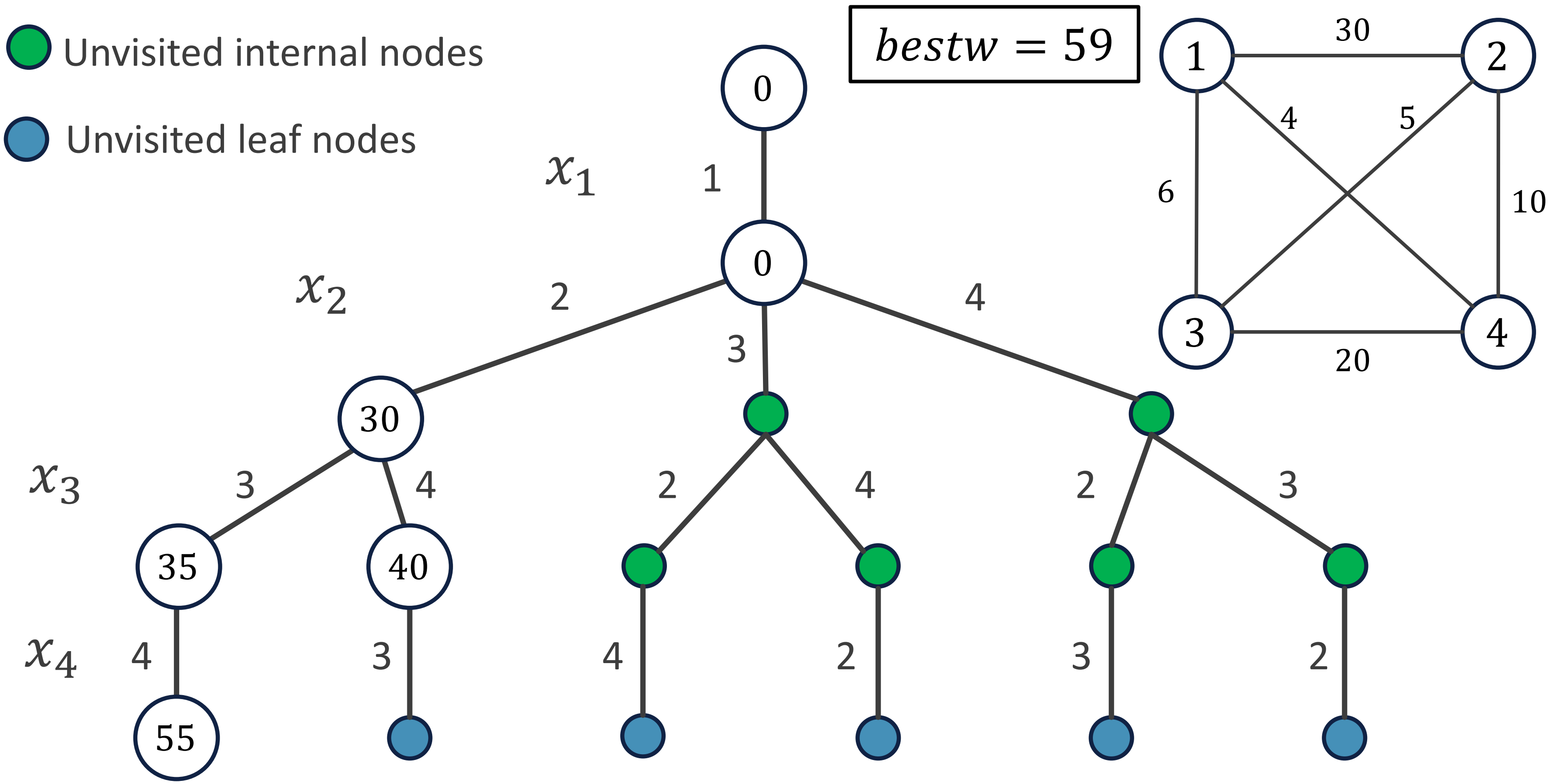


(1,2,3,4,1), 59

Example

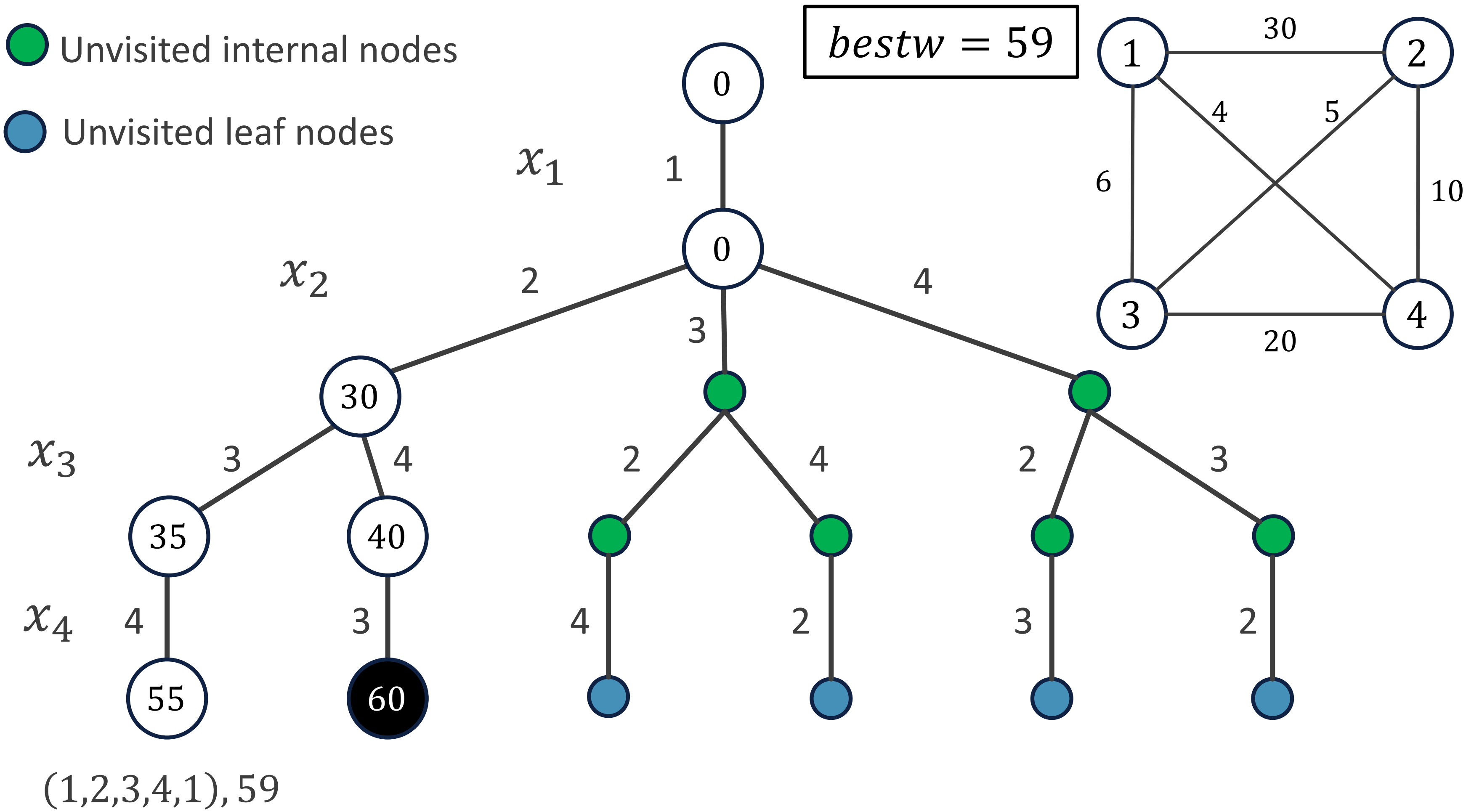
Unvisited internal nodes

Unvisited leaf nodes



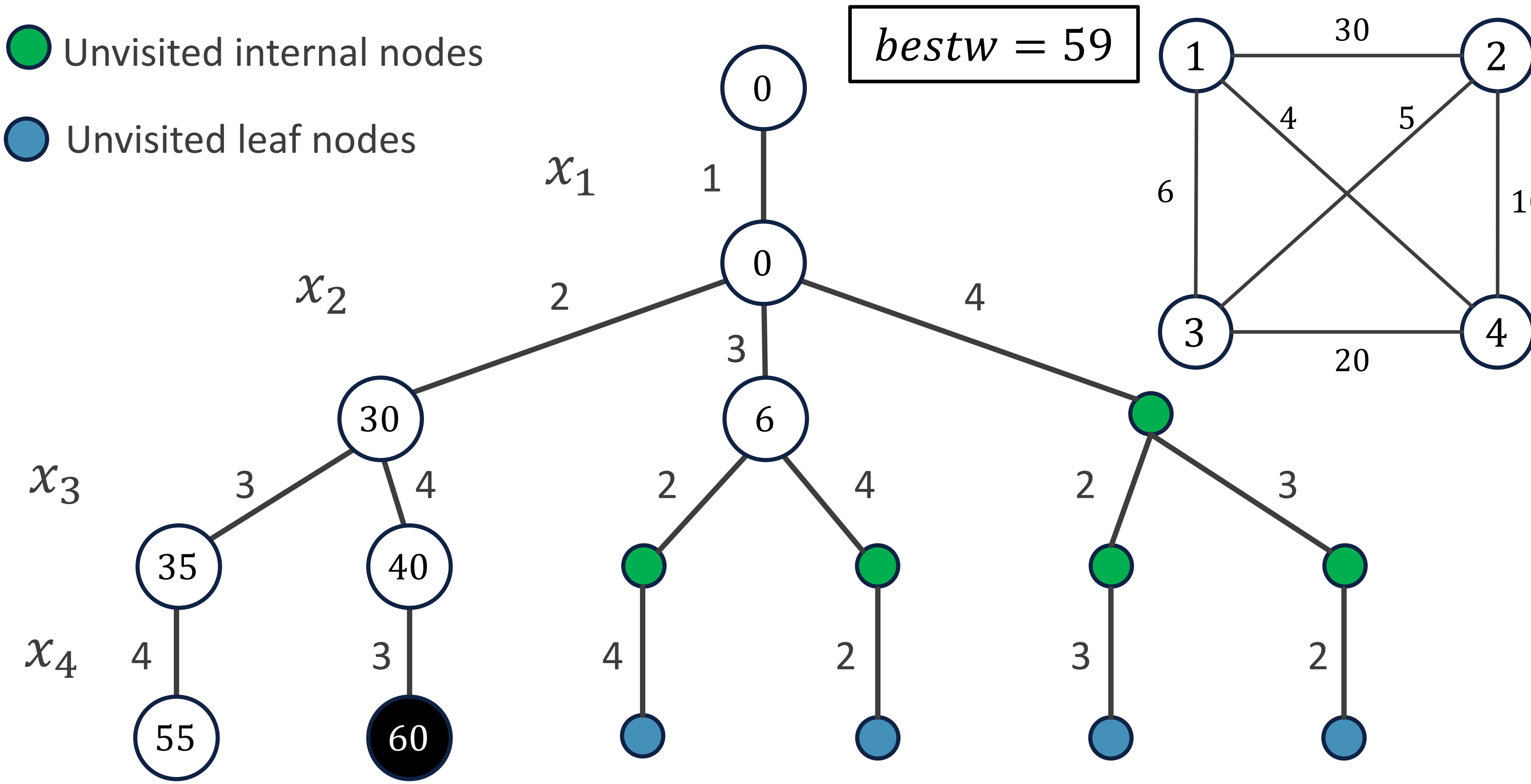
$(1,2,3,4,1), 59$

Example



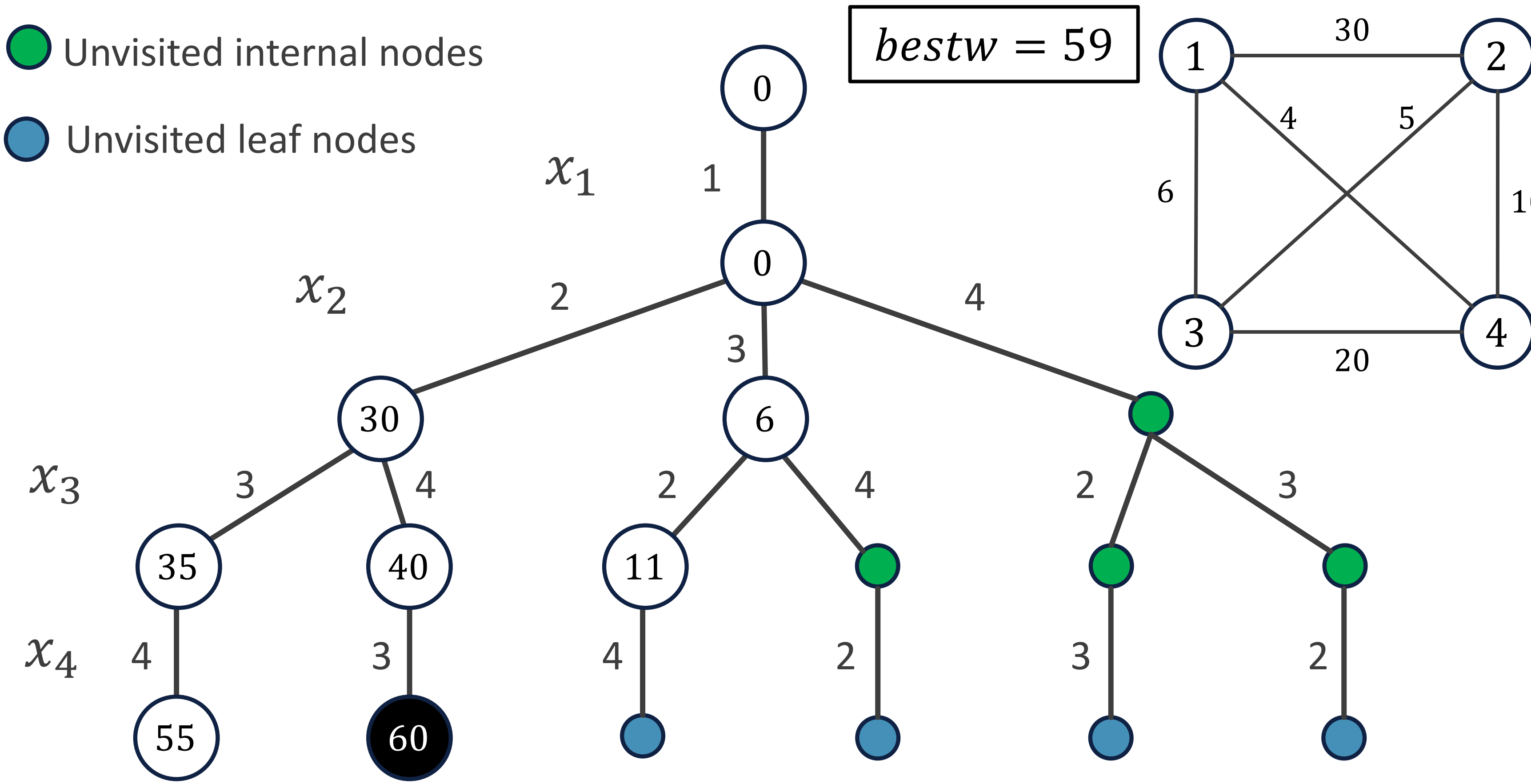
Unvisited internal nodes
Unvisited leaf nodes
$bestw = 59$
x_1
x_2
x_3
x_4
0
1
0
2
3
4
30
3
4
2
4
2
3
35
40
4
3
4
2
3
2
55
60
(1,2,3,4,1), 59
1
2
3
4
30
4
5
6
10
20

Example



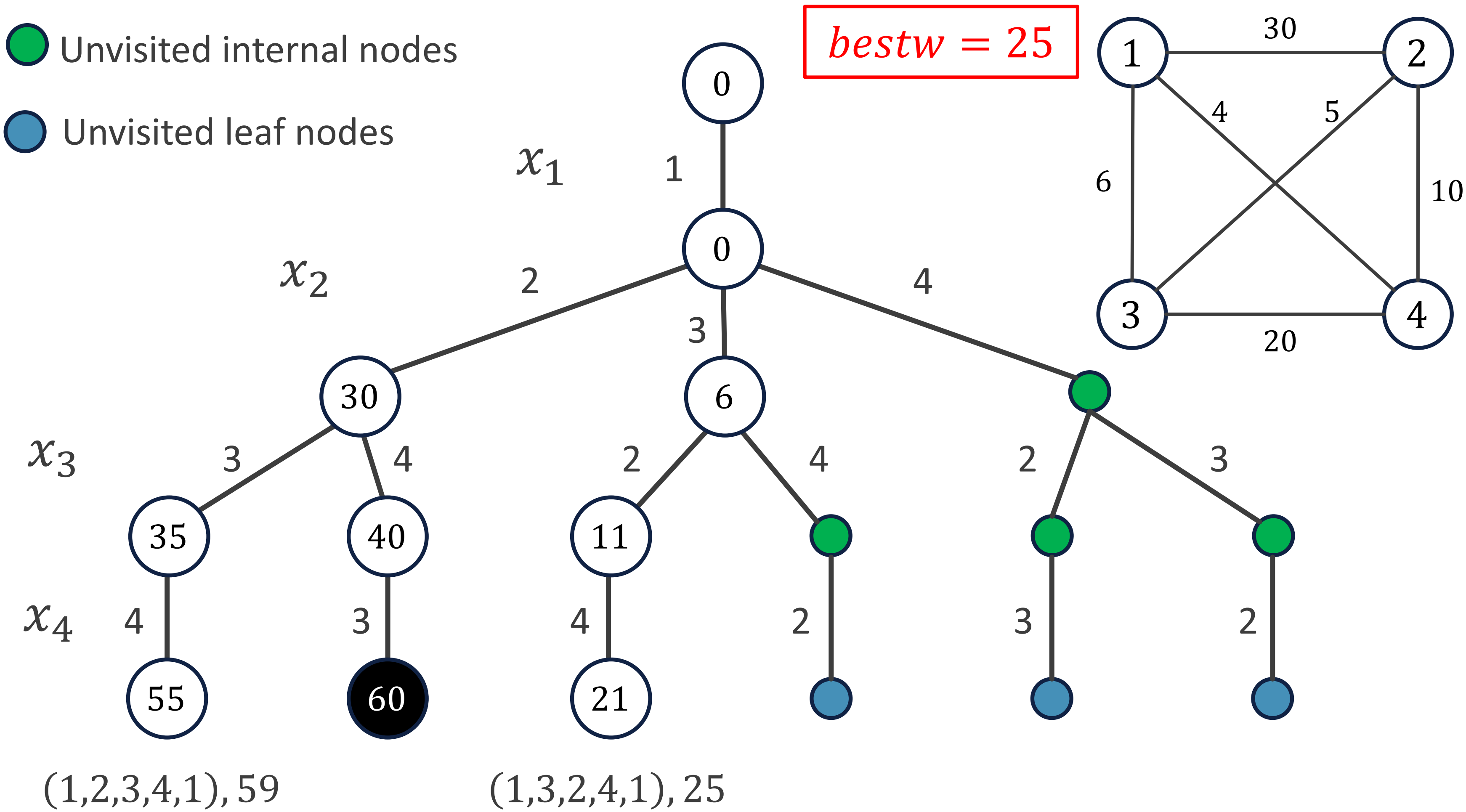
(1,2,3,4,1), 59

Example



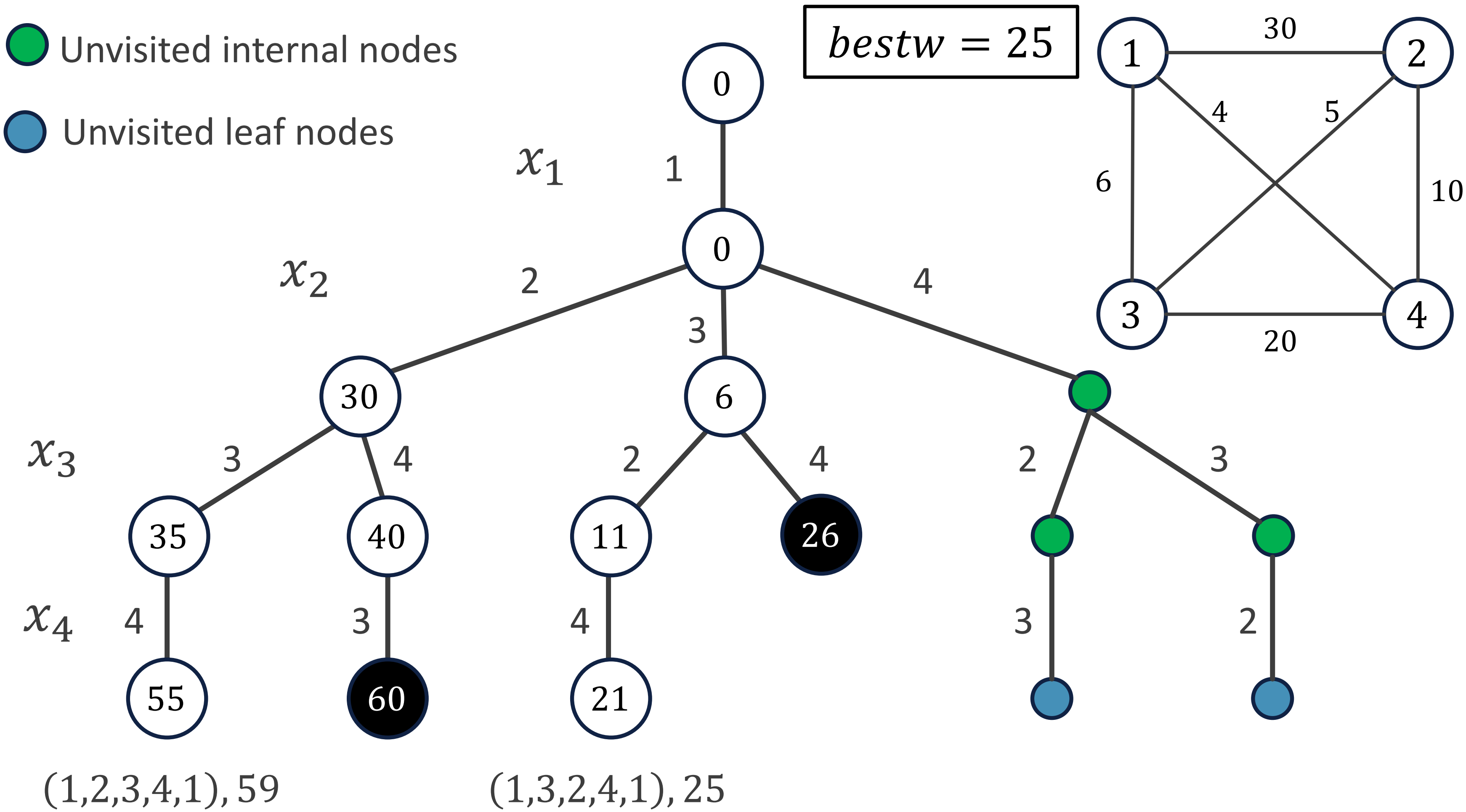
$(1,2,3,4,1), 59$

Example



Unvisited internal nodes
Unvisited leaf nodes
$bestw = 25$
x_1
x_2
x_3
x_4
(1,2,3,4,1), 59
(1,3,2,4,1), 25

Example



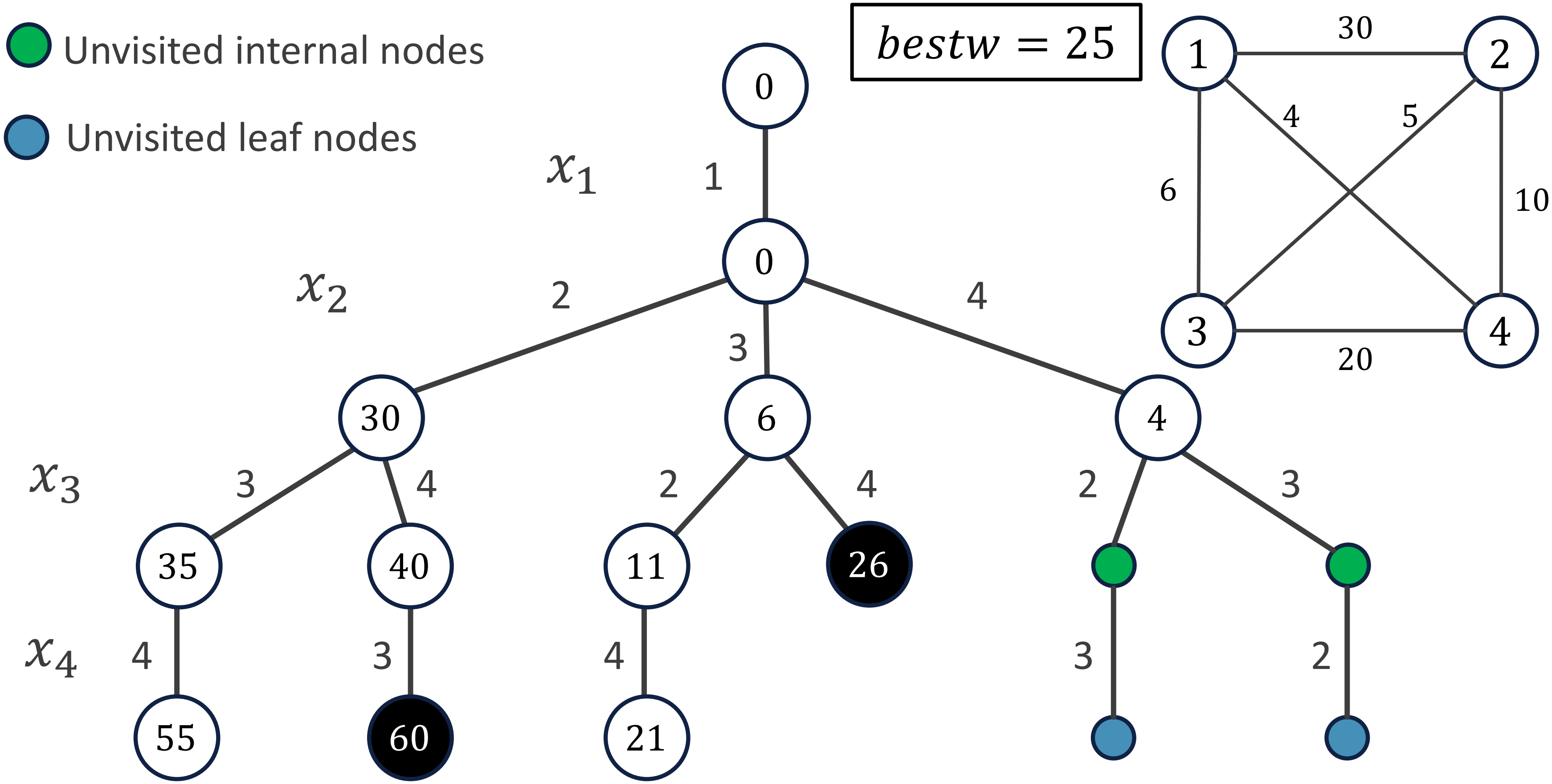
Unvisited internal nodes
Unvisited leaf nodes
bestw = 25
x_1
x_2
x_3
x_4
0
0
30
6
35
40
11
26
55
60
21
1
2
3
4
30
5
6
10
20
(1,2,3,4,1), 59
(1,3,2,4,1), 25

Example

Unvisited internal nodes

Unvisited leaf nodes

$bestw = 25$



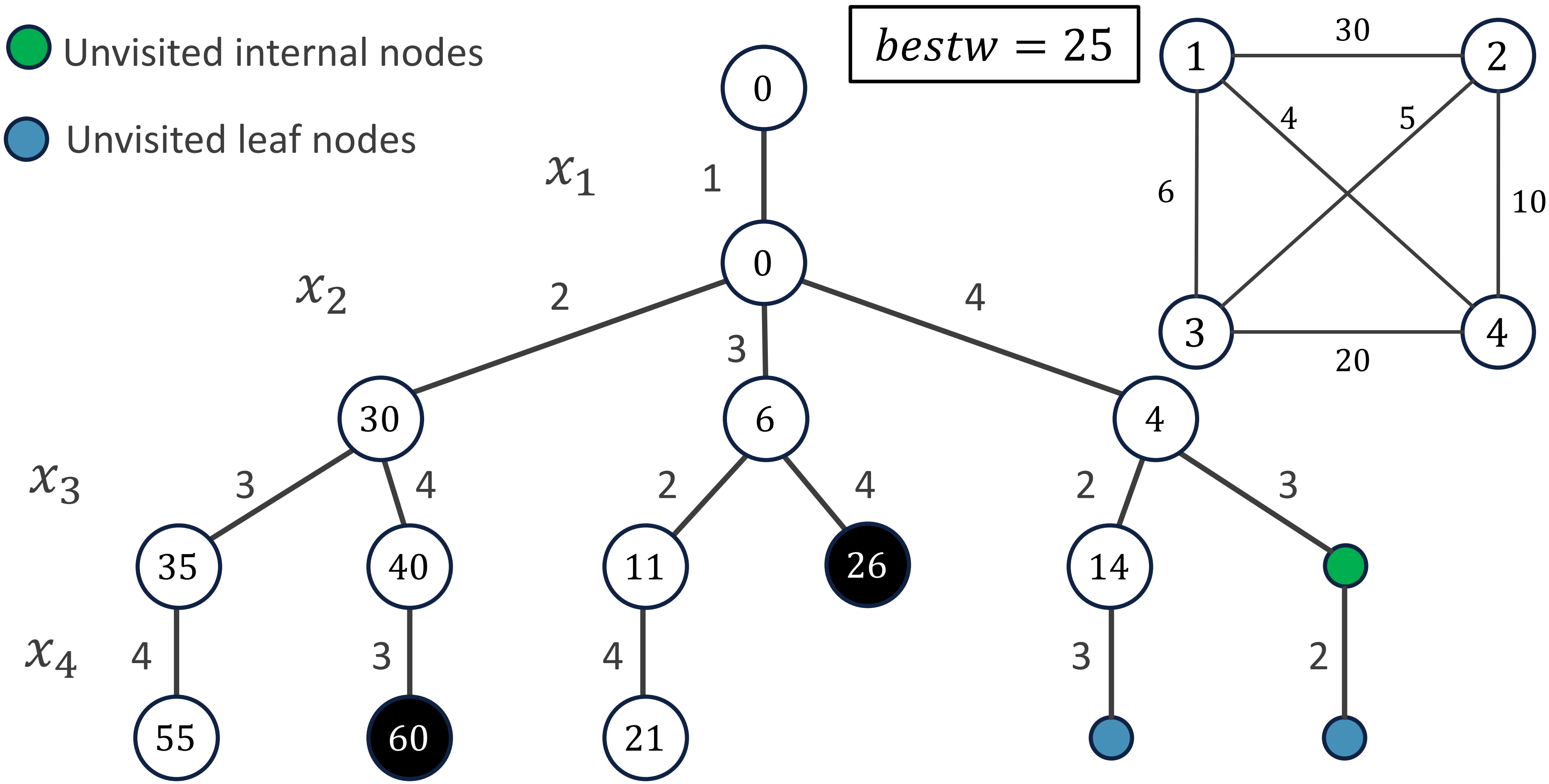
(1,2,3,4,1), 59

(1,3,2,4,1), 25

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系 Computer Science Department of Xiamen University

Example



(1,2,3,4,1), 59

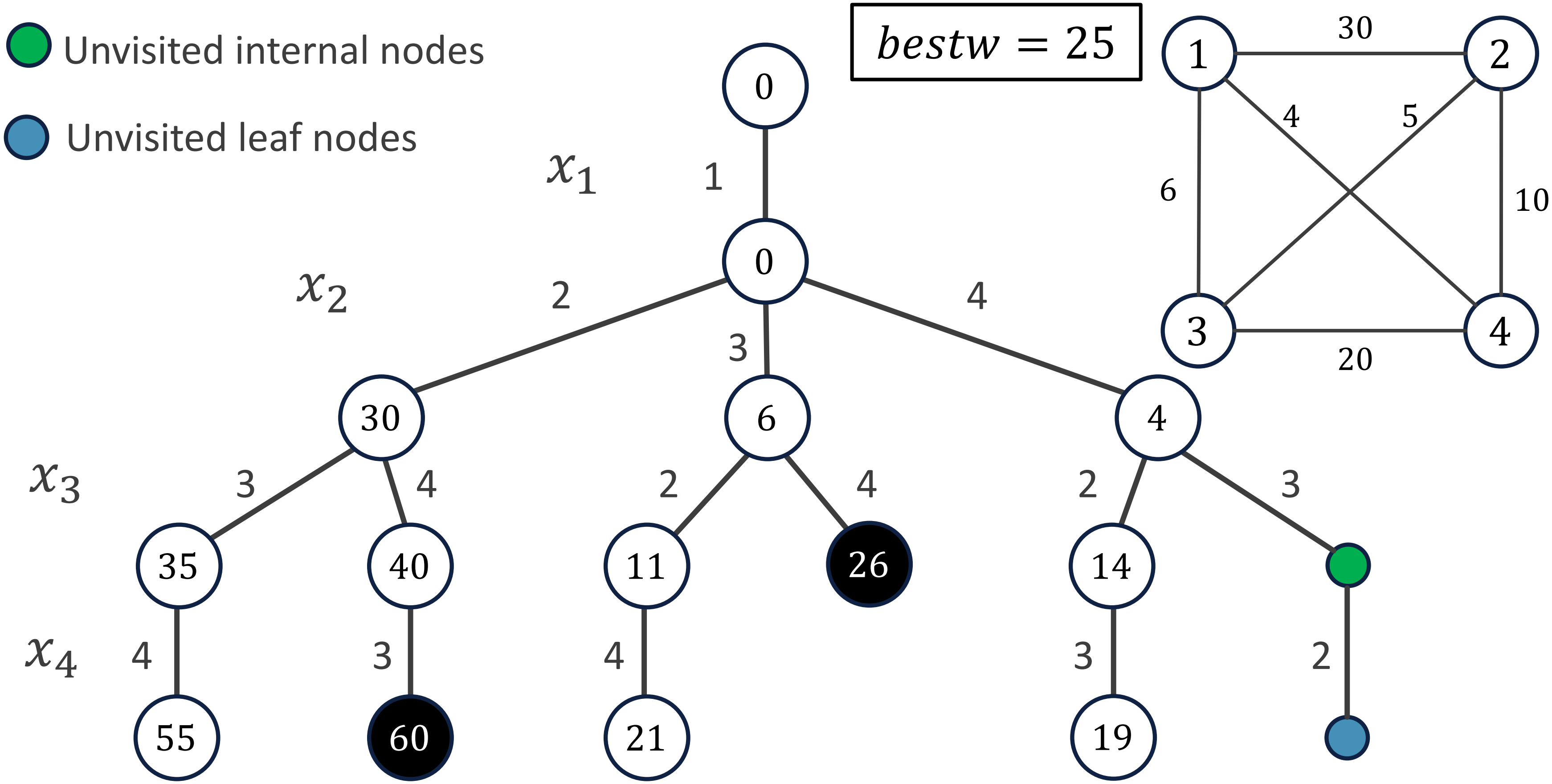
(1,3,2,4,1), 25

Example

Unvisited internal nodes

Unvisited leaf nodes

$bestw = 25$



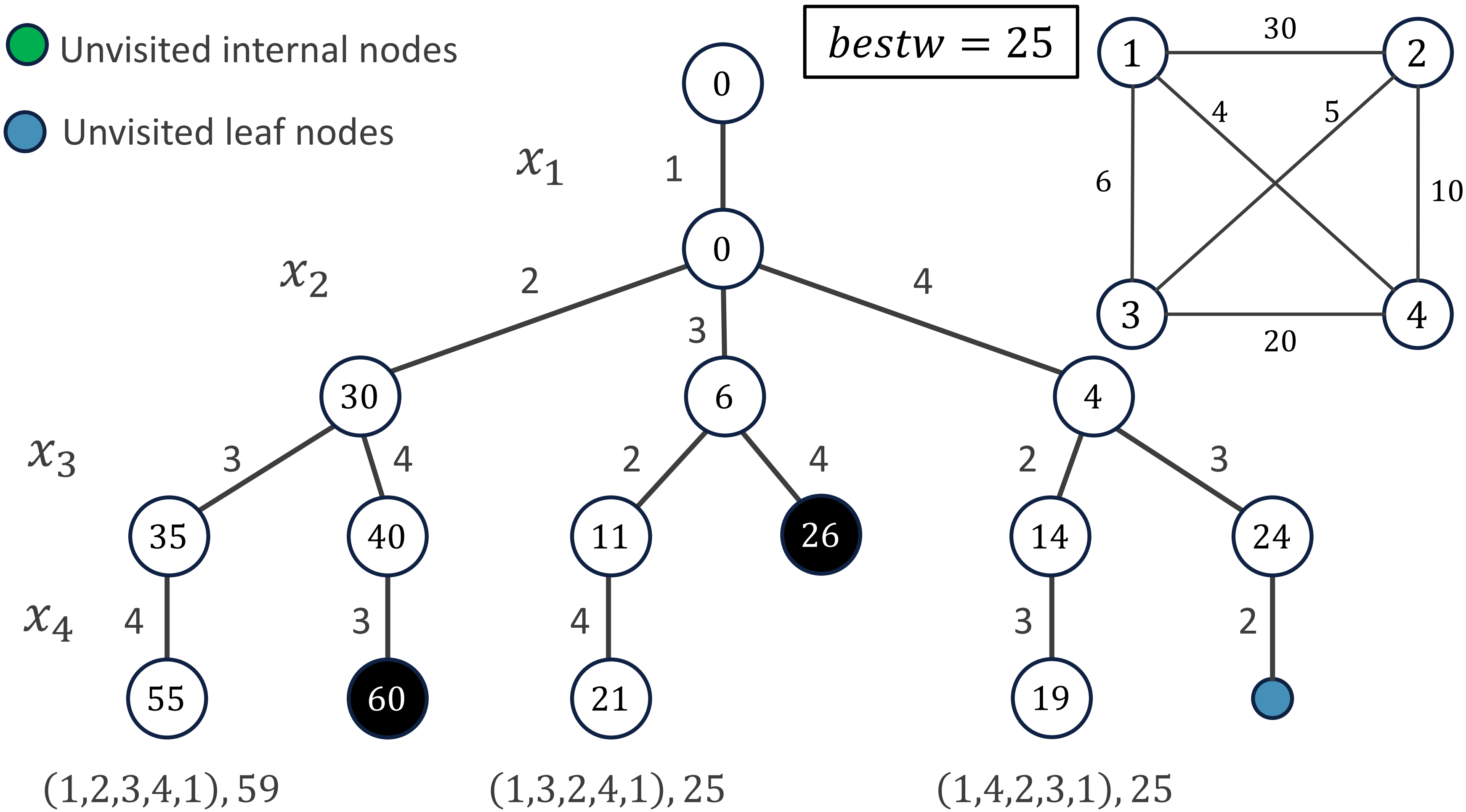
(1,2,3,4,1), 59

(1,3,2,4,1), 25

(1,4,2,3,1), 25

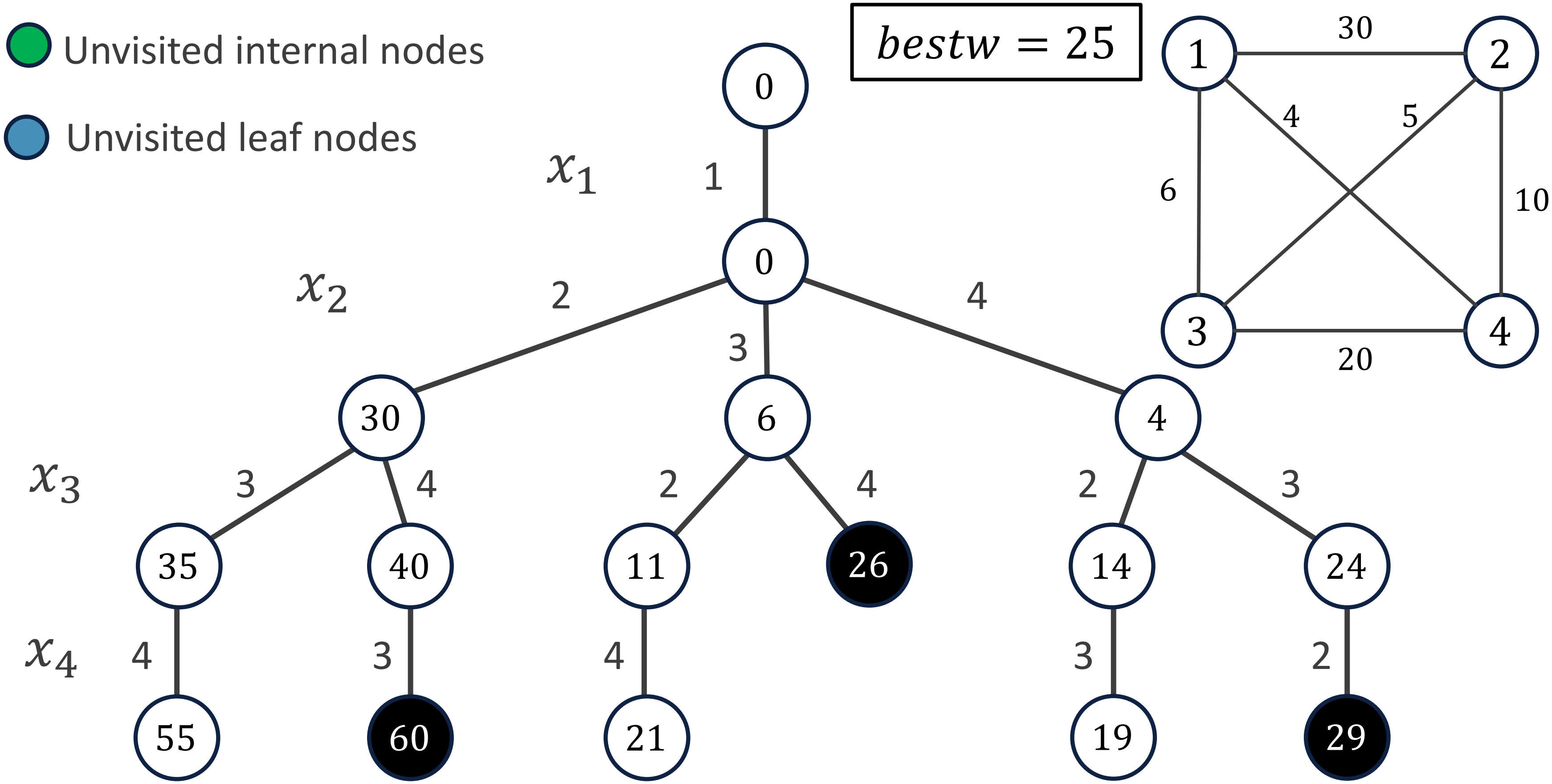
Example

- Unvisited internal nodes
- Unvisited leaf nodes



Example

- Unvisited internal nodes
- Unvisited leaf nodes



(1,2,3,4,1), 59 (1,3,2,4,1), 25 (1,4,2,3,1), 25

Pseudocode

Call BacktrackTSP(2) with initialization $x[i] = i$.

Connectivity between the last two vertices

Connectivity between the last and the first vertex

BacktrackTSP(i)

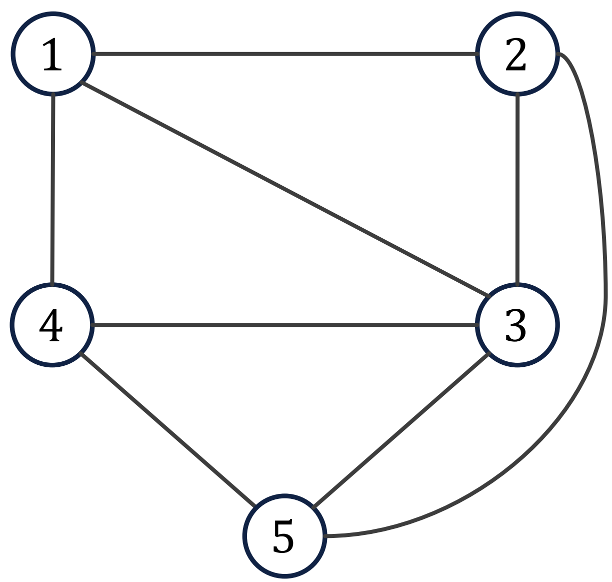
if $i = n$ **then**
    if $w[x[n-1], x[n]] \neq \infty$ and $w[x[n], 1] \neq \infty$ **then**
        if $cw + w[x[n-1], x[n]] + w[x[n], 1] < bestw$ **then**
            $bestw \leftarrow cw + w[x[n-1], x[n]] + w[x[n], 1]$
            for $j \leftarrow 1$ **to** n **do**
                $bestx[j] \leftarrow x[j]$
else for $j \leftarrow i$ **to** $n-1$ **do**
    if $w[x[i-1], x[j]] \neq \infty$ and $cw + w[x[i-1], x[j]] < bestw$ **then**
            $x[i] \leftrightarrow x[j]$
            $cw \leftarrow cw + w[x[i-1], x[i]]$
            BacktrackTSP($i + 1$)
            $cw \leftarrow cw - w[x[i-1], x[i]]$
            $x[i] \leftrightarrow x[j]$

We don't iterate to n because the last vertex is the only choice

We don't assign values to $x[i]$, instead we use permutation trick.

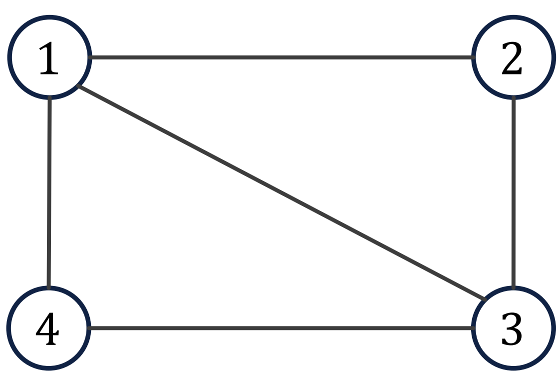
Classroom Exercise

Consider the 3-coloring problem for the given graph. Design constraint function and bounding function, and draw the pruned solution space tree to find a solution.



Classroom Exercise

- The constraint function is to check duplicated color.
- There is no bounding function for m-coloring problem.



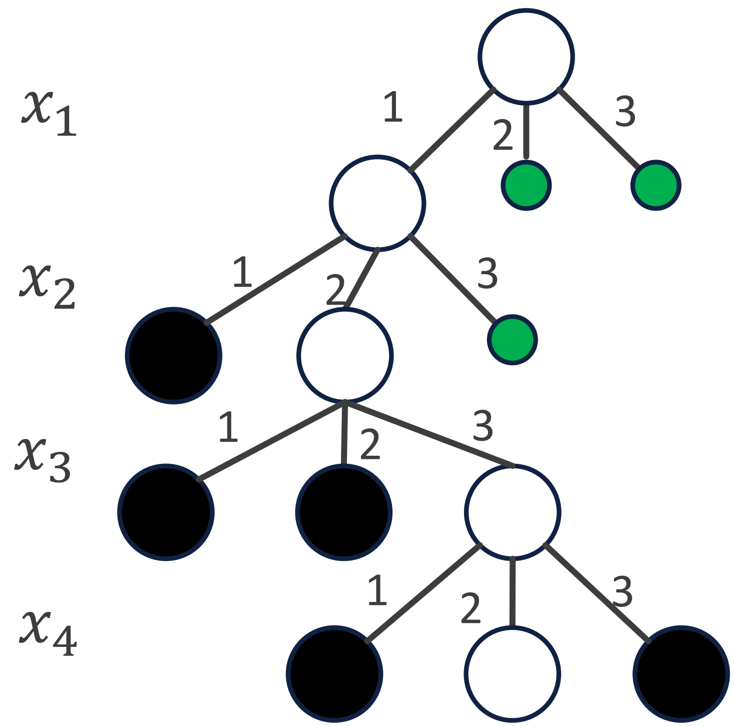


Image source: Figure 5.12, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Conclusion

After this lecture, you should know:

- What is the difference between DFS and backtracking.
- What is a solution space tree.
- What is constraint function and bounding function.
- What kind of problems can be solved by backtracking.

Homework

Page 238-240

12.7

12.8

12.10

- For these questions, you should describe the idea of how to design constraint function and bounding function. And then write down the pseudocode.

Experiment 1

- Write a program to solve a Sudoku puzzle by filling the empty cells.
- A sudoku solution must satisfy all of the following rules:
  - Each of the digits 1-9 must occur exactly once in each row.
  - Each of the digits 1-9 must occur exactly once in each column.
  - Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.
- Empty cells are indicated by the character '.'.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Experiment 1

- Input: [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Experiment 2

- 使用回溯解决石材切割问题.

谢谢

有问题欢迎随时跟我讨论

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系
Computer Science Department of Xiamen University