COT 6405
ANLYSIS OF ALGORITHMS

Backtracking
and
Branch-and-Bound

Computer & Electrical Engineering and Computer Science Dept.
Florida Atlantic University

Spring 2017

Backtracking and Branch-and-Bound

- Both considered improvements over exhaustive search
- Construct candidate solutions one component at a time and evaluate the partially constructed solutions
  - If no potential values of the remaining components can lead to a solution, the remaining components are not generated at all
- Based on the construction of a **search tree**; nodes reflect specific choices made for solution's components
  - Terminate a node when no solution to the problem can be obtained using node's descendants

Backtracking and Branch-and-Bound

- Different types of problems:
  - *Branch-and-bound* applicable to optimization problems; based on computing a bound on possible values of the problem's objective function
  - *Backtracking* more often applied to non-optimization problems
- Order in which nodes of the search tree are generated
  - *Backtracking*: tree usually developed depth first (like DFS)
  - *Branch-and-bound* generates nodes using several rules; most natural one is the *best-first* rule

Backtracking

Search tree:

- The root – initial state before the search for a solutions begins
- Nodes on the first level – choices made for the first component of a solution
- Nodes on the second level – choices for the second component
- So on …
- A node is **promising** if it corresponds to a partially constructed solution that may still lead to a full solution; otherwise it is **nonpromising**
- Leaves are either nonpromising dead ends or complete solutions
- Constructed in a manner of depth-first-search

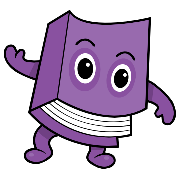
Backtracking

Search tree construction:

- If the current node is *promising*, its child is generated by adding the first remaining legitimate option for the next component of a solution
  - Then the process moves to this child
- If the current node is *nonpromising*, then the algorithm backtracks to the node's parent to consider the next possible option for its last component
  - If no such option, then backtrack one more level up in the tree, and so on
- If a complete solution is found, then the algorithm either stops (if only one solution is required) or continues searching for other possible solutions

Problems discussed in class

- *n*-queens problem
- The Hamiltonian Cycle (HC) problem



Reading assignment:

- Backtracking chapter posted on Canvas

Branch-and-Bound

- Branch-and-bound technique
- Methods to construct the search tree:
  - Breadth-First-Search (BreadthFS)
  - Best-First-Search (BestFS)
- Two problems:
  - Knapsack Problem
  - Traveling Salesman Problem (TSP)

Branch-and-Bound

- ***Optimization problem*** – problem that seeks to minimize or maximize an objective function
- ***Feasible solution*** – point in the problem's search space that satisfies the problem's constraints
- ***Optimal solution*** – feasible solution with the best value of the objective function
- Branch-and-bound is used for optimization problems

Branch-and-Bound

- Compared to backtracking, branch-and-bound requires two additional items:
  - A way to provide, for every node in the search tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
  - The value of the best solution so far

Branch-and-Bound Technique

- Basic idea: a node is ***nonpromising*** (e.g. the branch is *pruned*) if the node bound value is not better than the best solution seen so far:
  - Not smaller for a minimization problem
  - Not larger for a maximization problem

Branch-and-Bound Technique

A search path terminates at the current node for one of the following three reasons:

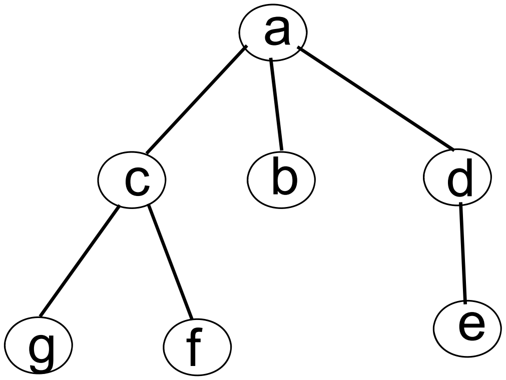
1. The value of the node's bound is not better than the value of the best solution seen so far
2. The node represents no feasible solution
3. This node represents a feasible solution; compare the value of its objective function with the value of the best solution seen so far; if the new solution is better, then update the best solution seen so far

Branch-and-Bound

- How to generate nodes in the search tree?
  - Breadth-first-search (**BreadthFS**) with branch-and-bound pruning
  - Best-first-search (**BestFS**) with branch-and-bound pruning

Breadth-first-search (BreadthFS) - REVIEW

```
BreadthFS(T)
Q = ∅
r = T.root
ENQUEUE (Q, r)
visit r
while Q ≠ ∅
    v = DEQUEUE(Q)
     for each child u of v
         visit u
         ENQUEUE (Q,u)
```



- BreadthFS has RT = $\Theta(V+E)$
- since G is a tree, RT = $\Theta(V)$

Knapsack Problem

- Given n items:
  - weights: w_1 w_2 … w_n
  - profit: p_1 p_2 … p_n
  - a knapsack of capacity W
- Find most valuable subset of items that fit into the knapsack

- Example: Knapsack capacity W=16

item	weight	profit	p_i/w_i
1	2	\$40	\$20
2	5	\$30	\$6
3	10	\$50	\$5
4	5	\$10	\$2

BreadthFS w/ Branch-and-bound pruning

- ***weight***, ***profit*** – total weight and total profit of the items that have been included up to a node
- compute an upperbound at each node: greedily grab items until ***totalweight*** > W
- assume current node is at level *i* and that item *k* would bring the weight above W:

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (W - totweight) \times \frac{p_k}{w_k}$$

upperbound on the profit of the current partial solution

General Algorithm for BreadthFS with Branch-and-Bound

```
BreadthFS-Branch-and-Bound(T, best)
Q = ∅
r = T.root
ENQUEUE(Q,r)
best = value(r)
while Q ≠ ∅
    v = DEQUEUE(Q)
    for each child u of v
        if value(u) is better than best
            best = value(u)
        if bound(u) is better than best
            ENQUEUE(Q,u)
```

- *bound( )* and *value( )* are application dependent

Knapsack problem

- Each node is an object with fields:

    v.level – node's level in the tree

    v.profit

    v.weight

Knapsack with BreadthFS w/ Branch-and-Bound pruning(n,p[ ],w[ ],W,maxprofit)

```
Q = ∅
r.level = 0; r.profit = 0; r.weight = 0
maxprofit = 0
ENQUEUE(Q,r)
while Q ≠ ∅
   v = DEQUEUE(Q)
   u.level = v.level +1
   u.weight = v.weight + w[u.level]
   u.profit = v.profit + p[u.level]
   if (u.weight ≤ W and u.profit > maxprofit)
           maxprofit = u.profit
   if bound(u) > maxprofit
           ENQUEUE(Q,u)
   u.weight = v.weight
   u.profit = v.profit
   if bound(u) > maxprofit
           ENQUEUE(Q,u)
```

set u as the child of v that includes the next item

set u as the child of v that does not include the next item

Knapsack with BreadthFS w/ Branch-and-bound pruning

```
bound(u)
if u.weight ≥ W
  return 0
else
  result = u.profit
  totweight = u.weight
  j = u.level + 1
  while (j ≤ n) and (totweight + w[j] ≤ W)
        totweight = totweight + w[j]
        result = result + p[j]
        j = j+1
  k = j
  if k ≤ n
        result = result + (W – totweight)× p_k/w_k
  return result
```

RT Analysis

- number of nodes:

$$\leq 1 + 2 + 2^2 + \ldots + 2^n = O(2^n)$$

- bound() takes O(n)

$\Rightarrow$ total RT = $O(n \cdot 2^n)$

BestFS w/ Branch-and-Bound Pruning

- Basic idea: when it comes to pick-up a new node in the search, choose the one w/ the best bound among all *promising* unexpanded nodes
- Often arrives at an optimal solution more quickly
  - there is no guarantee that the node that appears to be the best will actually lead to the optimal solution
- Example

BestFS w/ Branch-and-Bound Pruning

- Instead of using a queue, we use a *priority queue* PQ
- Operations:

  insert(PQ, v) – adds v to the PQ

  remove(PQ) – remove the node with the best bound

General Algorithm for BestFS with Branch-and-Bound

```
BestFS-Branch-and-Bound(T, best)
PQ = ∅
r = T.root
best = value(r)
insert(PQ,r)
while PQ ≠ ∅
   v = remove(PQ)
   if bound(v) is better than best
         for each child u of v
               if value(u) is better than best
                     best = value(u)
               if bound(u) is better than best
                     insert(PQ, u)
```

Knapsack problem

- Each node is an object with fields:
  - v.level – node's level in the tree
  - v.profit
  - v.weight
  - v.bound

Knapsack-BestFS-Branch-and-Bound(n,p[ ],w[ ],W,maxprofit)

```
PQ = ∅
r.level = r.profit = r.weight = 0
maxprofit = 0
r.bound = bound(r)
insert(PQ,r)
while PQ ≠ ∅
   v = remove(PQ)
   if v.bound > maxprofit
         u.level = v.level +1
         u.weight = v.weight + w[u.level]
         u.profit = v.profit + p[u.level]
         if (u.weight ≤ W and u.profit > maxprofit)
                  maxprofit = u.profit
         u.bound = bound(u)
         if bound(u) > maxprofit
                  insert(PQ,u)
         u.weight = v.weight
         u.profit = v.profit
         u.bound = bound(u)
         if u.bound > maxprofit
                  insert(PQ,u)
```

set u as the child of v that includes the next item

set u as the child of v that does not include the next item

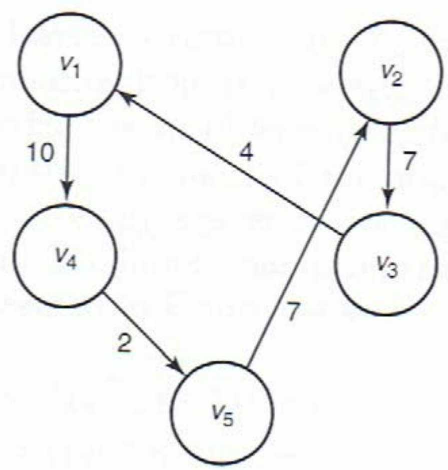
- function bound() is the same

Traveling Salesman Person (TSP)

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Example:

Adjacency matrix:

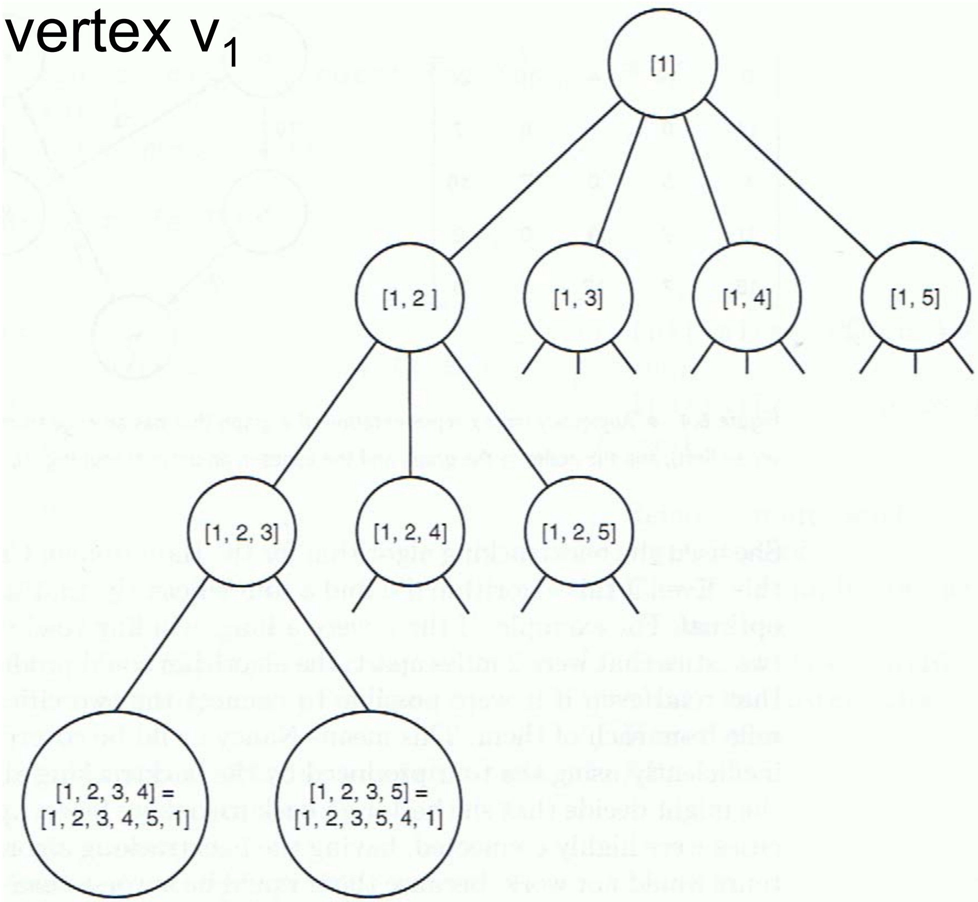
$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



min tour length = 30

TSP – search tree example

- Since the tour passes through all vertices, assume that it starts from the first vertex v_1



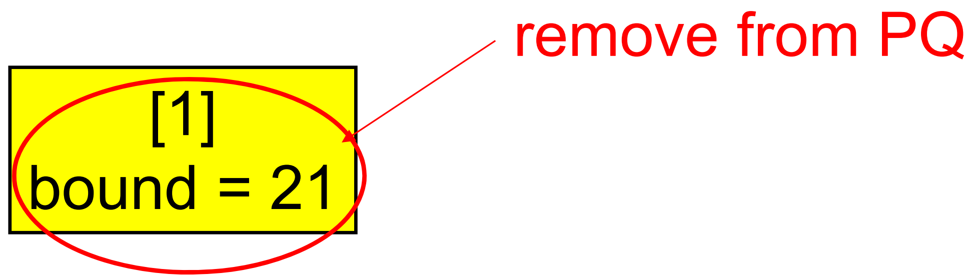
TSP with BestFS w/ Branch-and-bound pruning

- compute a bound for each node
- lower bound on the length of any tour that can be obtained by expanding beyond a given node
- any tour must leave each vertex exactly once, then a lower-bound is to take minimum edge leaving every vertex:

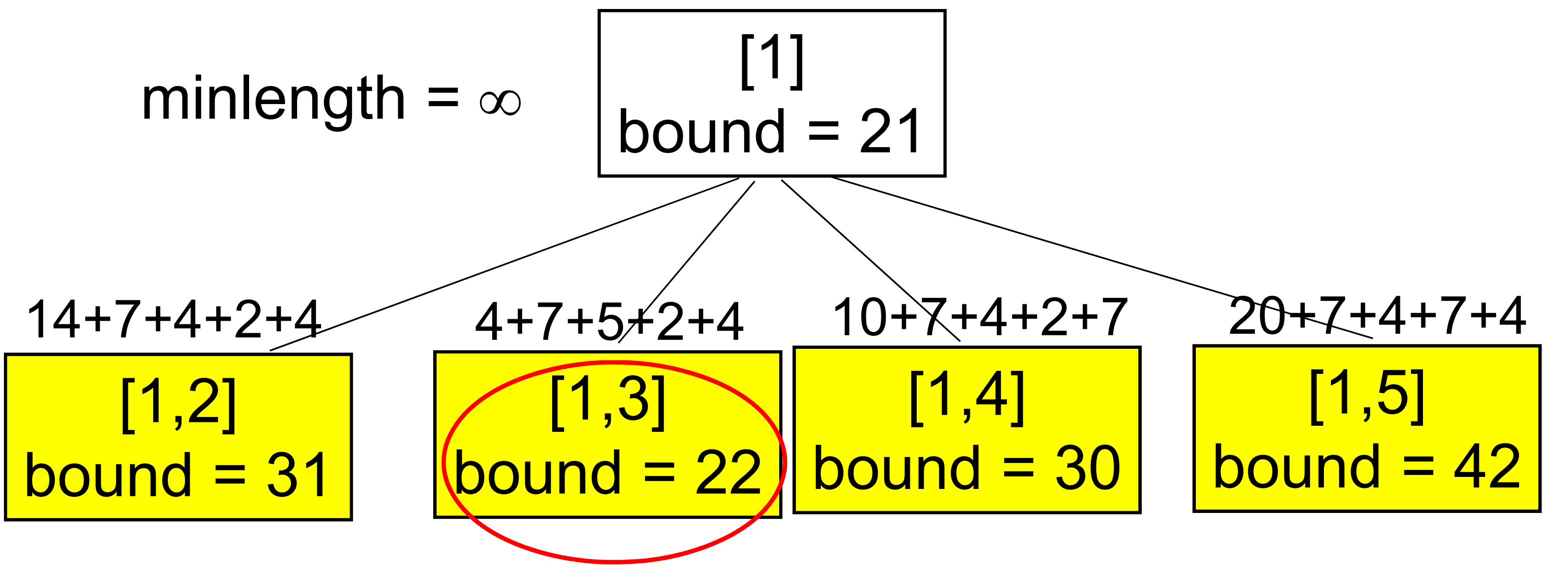
  v_1 minimum(14,4,10,20) = 4
  v_2 minimum(14,7,8,7) = 7
  v_3 minimum(4,5,7,16) = 4
  v_4 minimum(11,7,9,2) = 2
  v_5 minimum(18,7,17,4) = 4
- lower-bound on the length of a tour is 4+7+4+2+4 = 21
- Observation: this does not mean there is a tour w/ this length; it means there is no tour w/ a shorter length

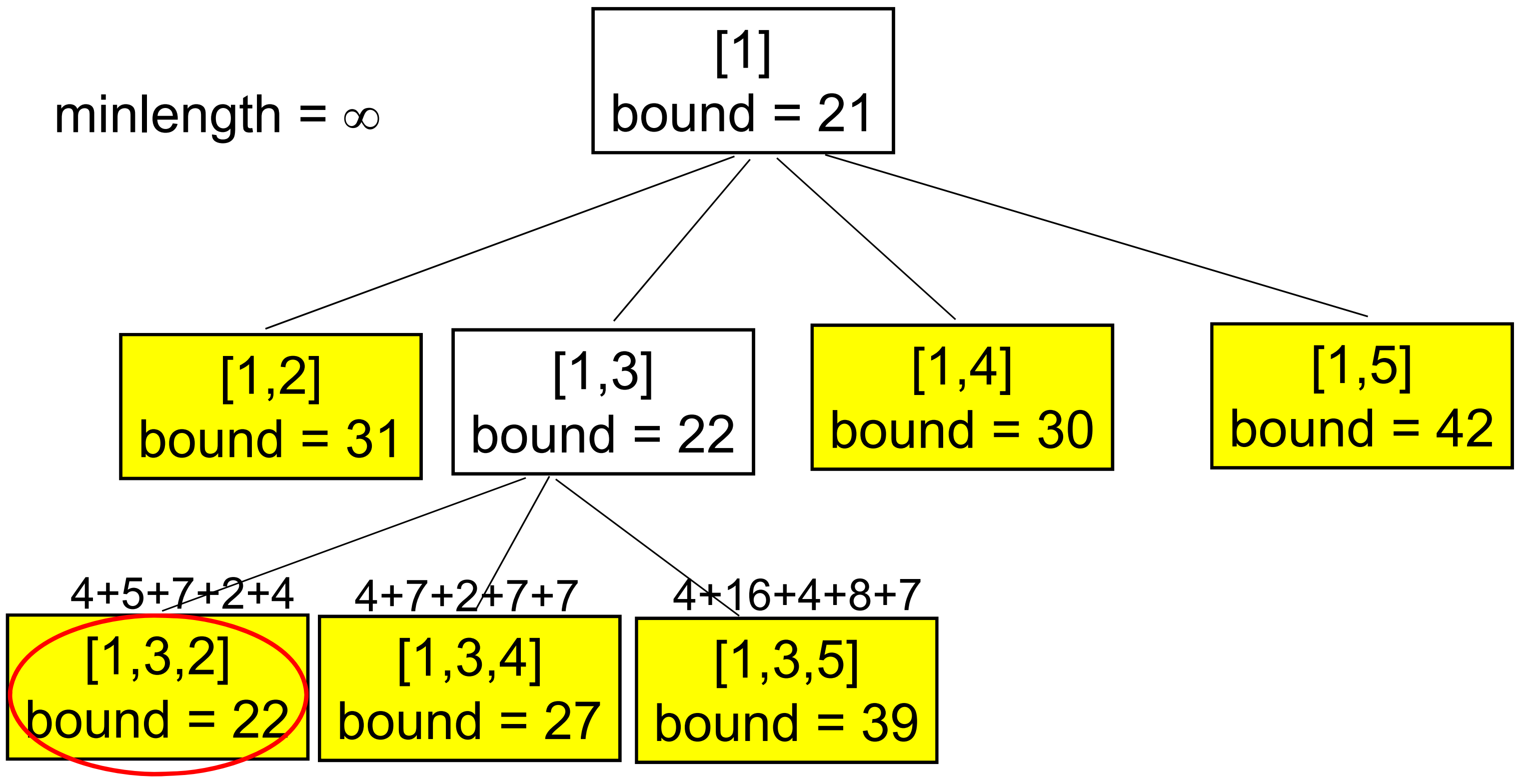
TSP Example

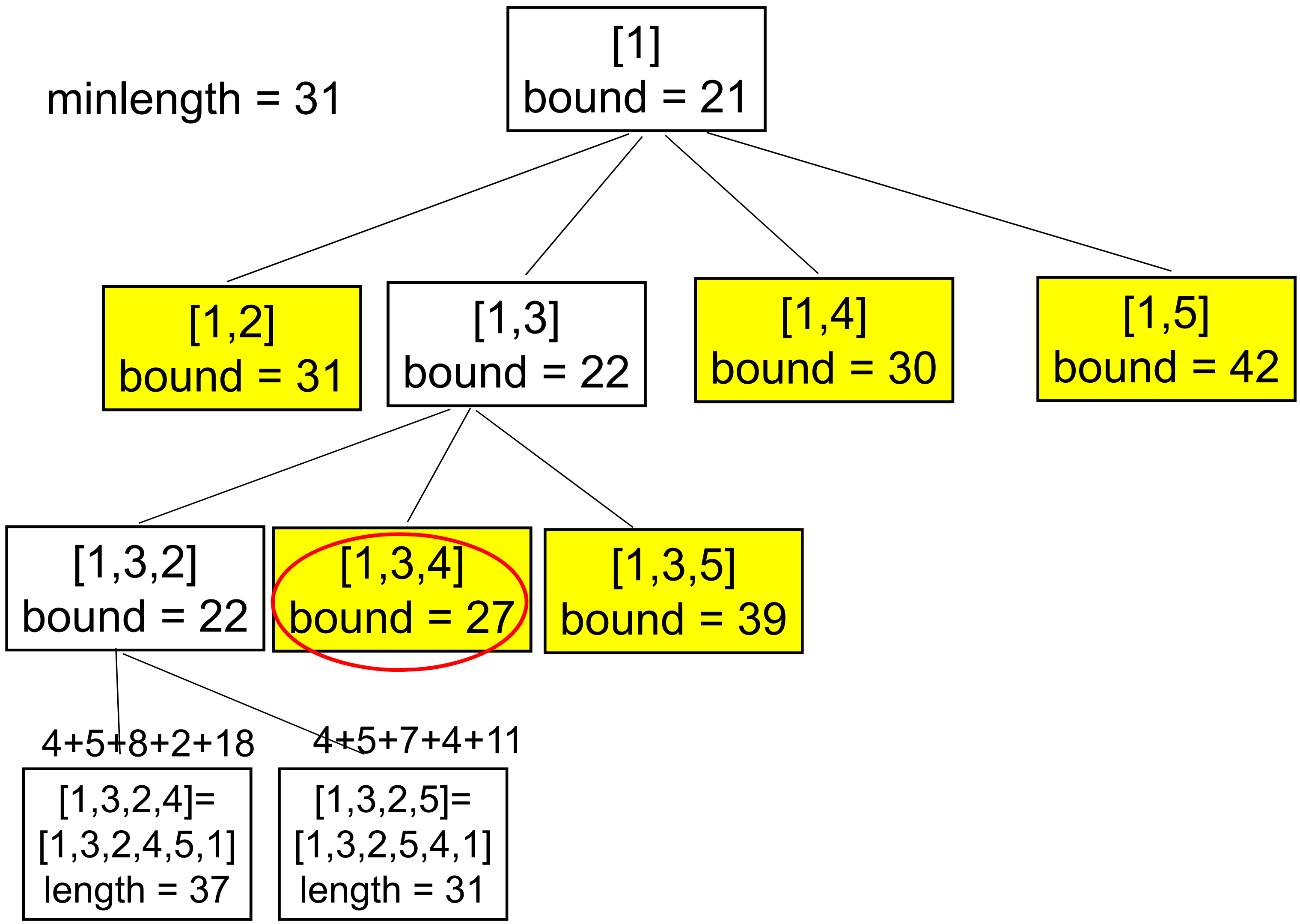
minlength = ∞

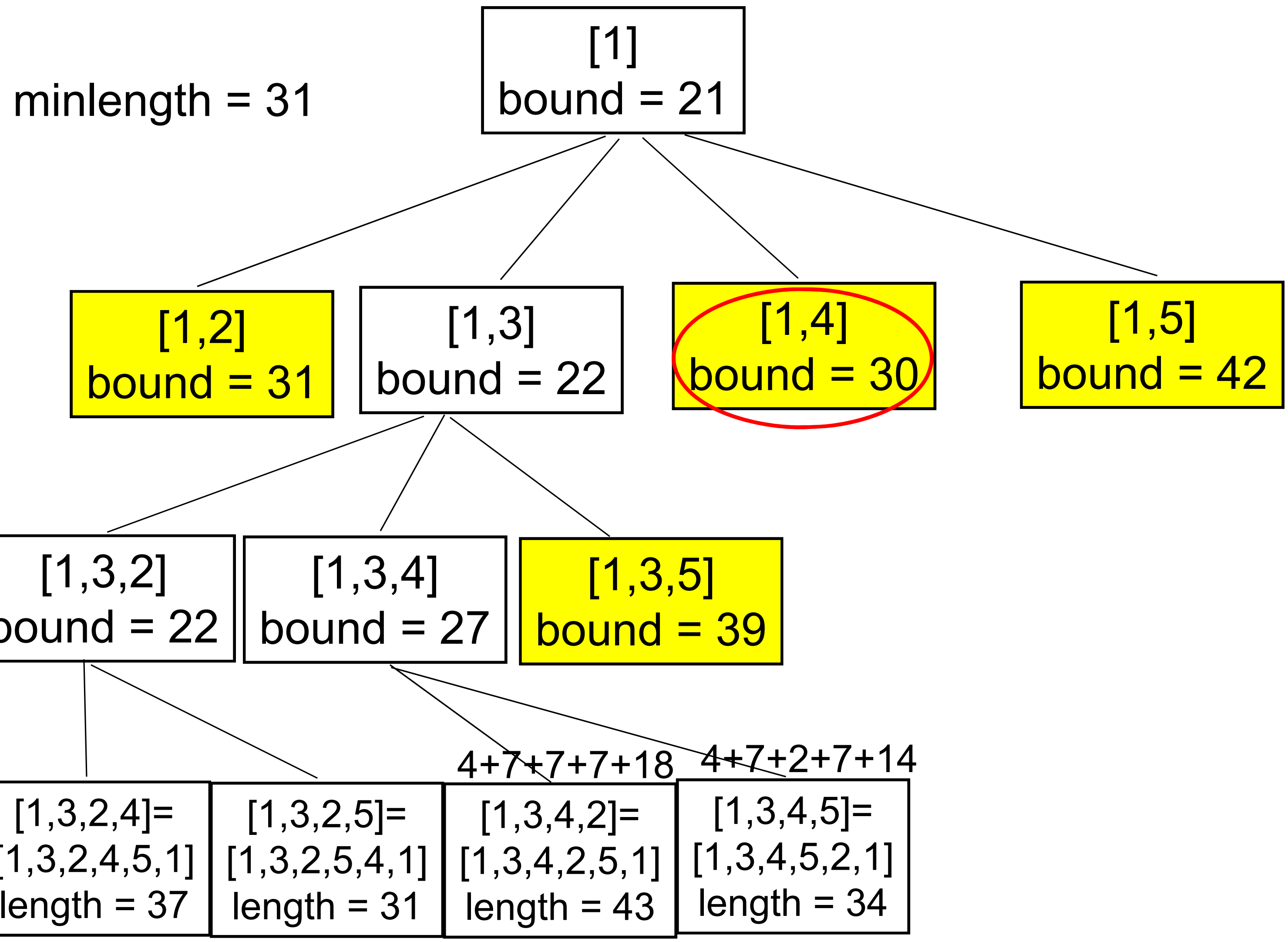


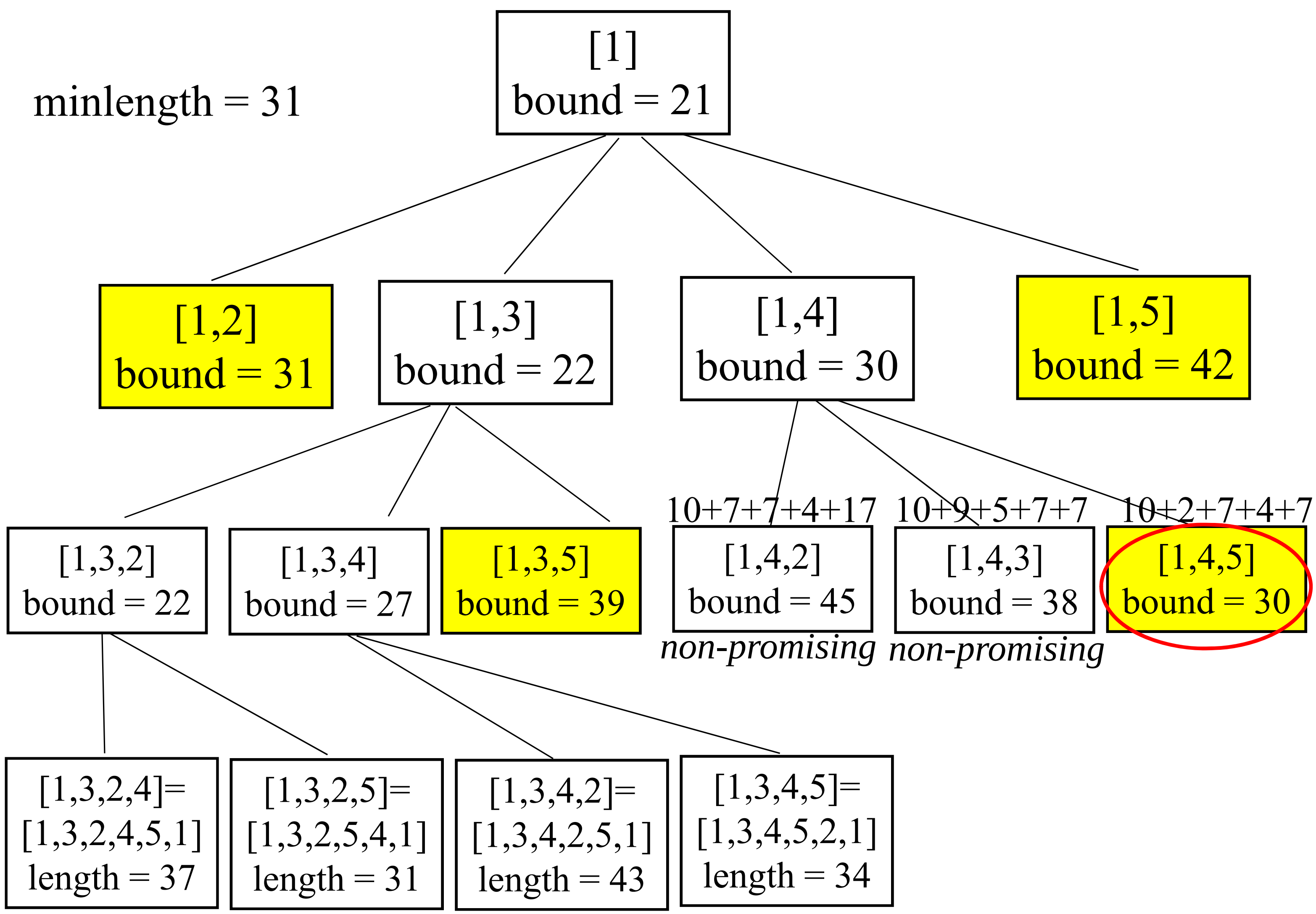
minlength = ∞

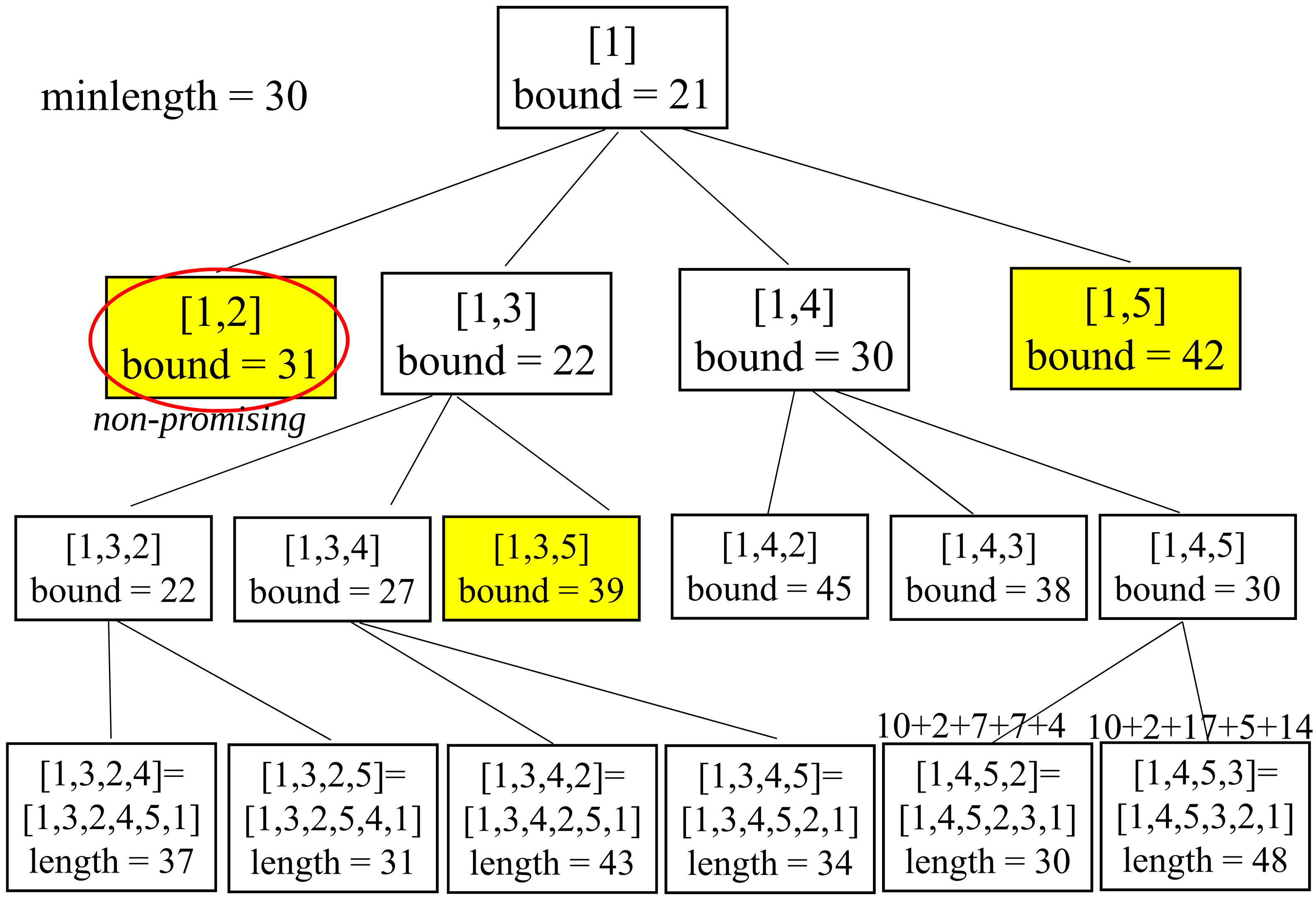


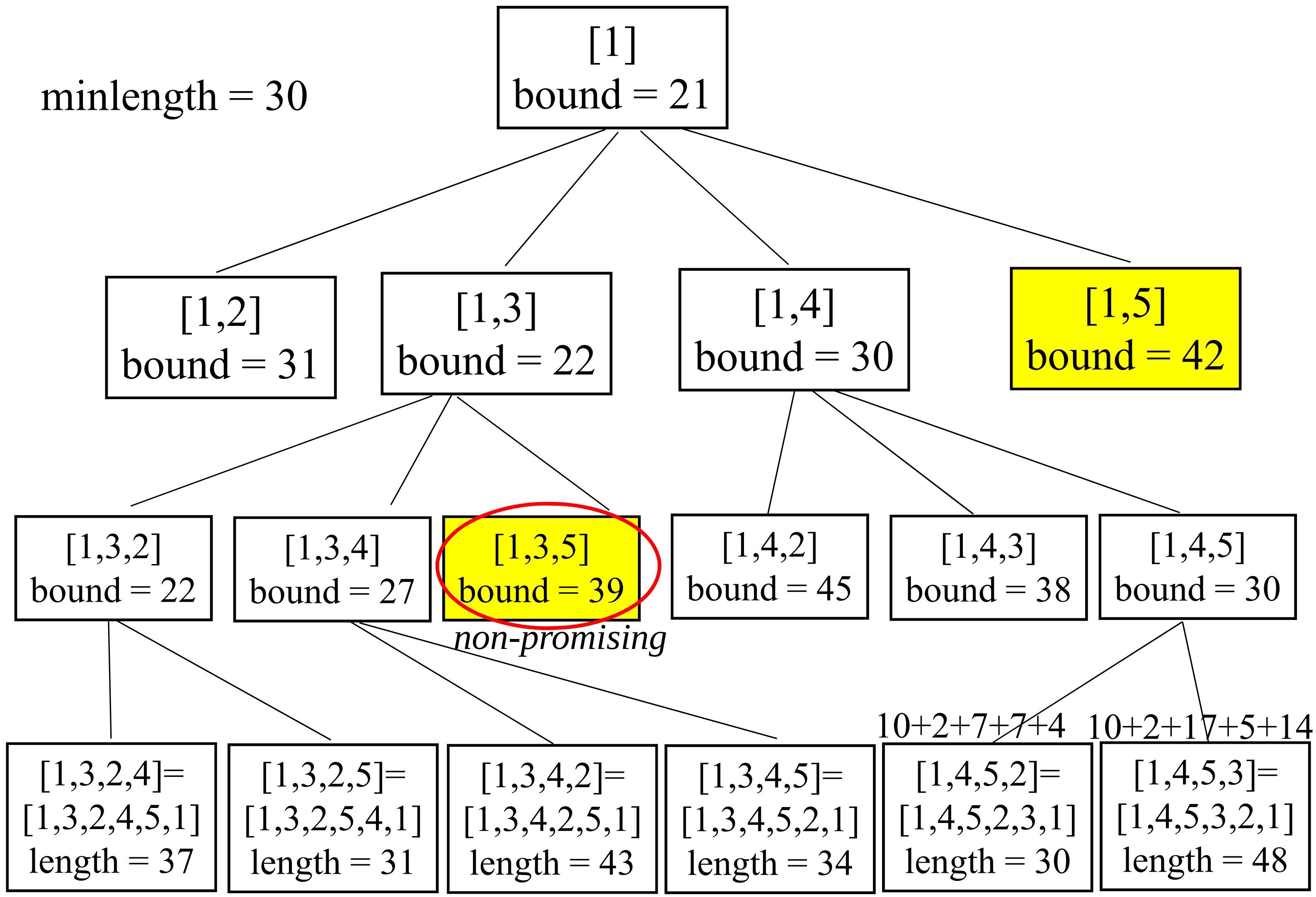

minlength = ∞
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
4+5+7+2+4
[1,3,2]
bound = 22
4+7+2+7+7
[1,3,4]
bound = 27
4+16+4+8+7
[1,3,5]
bound = 39

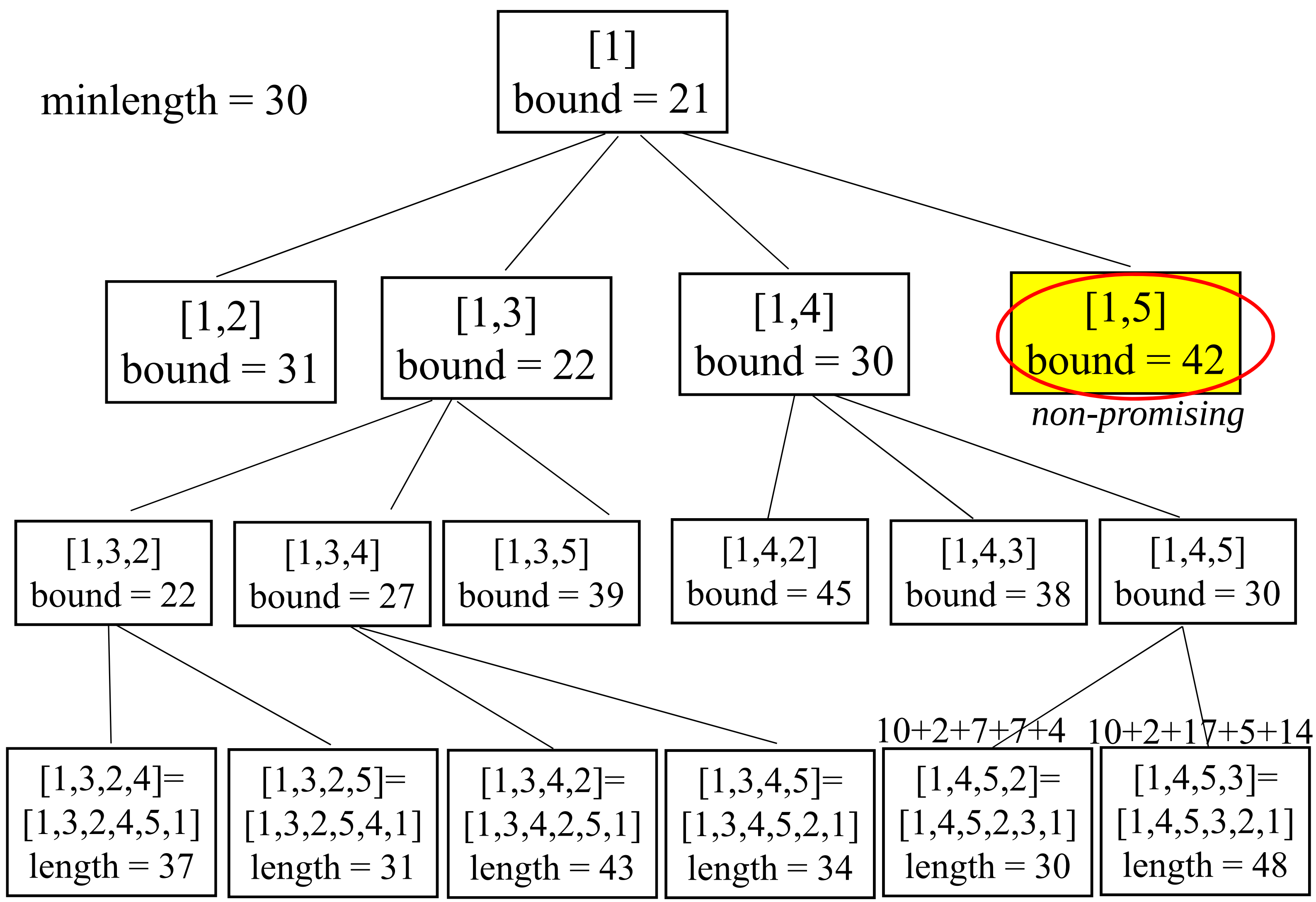

minlength = 31
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
4+5+8+2+18
4+5+7+4+11
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31

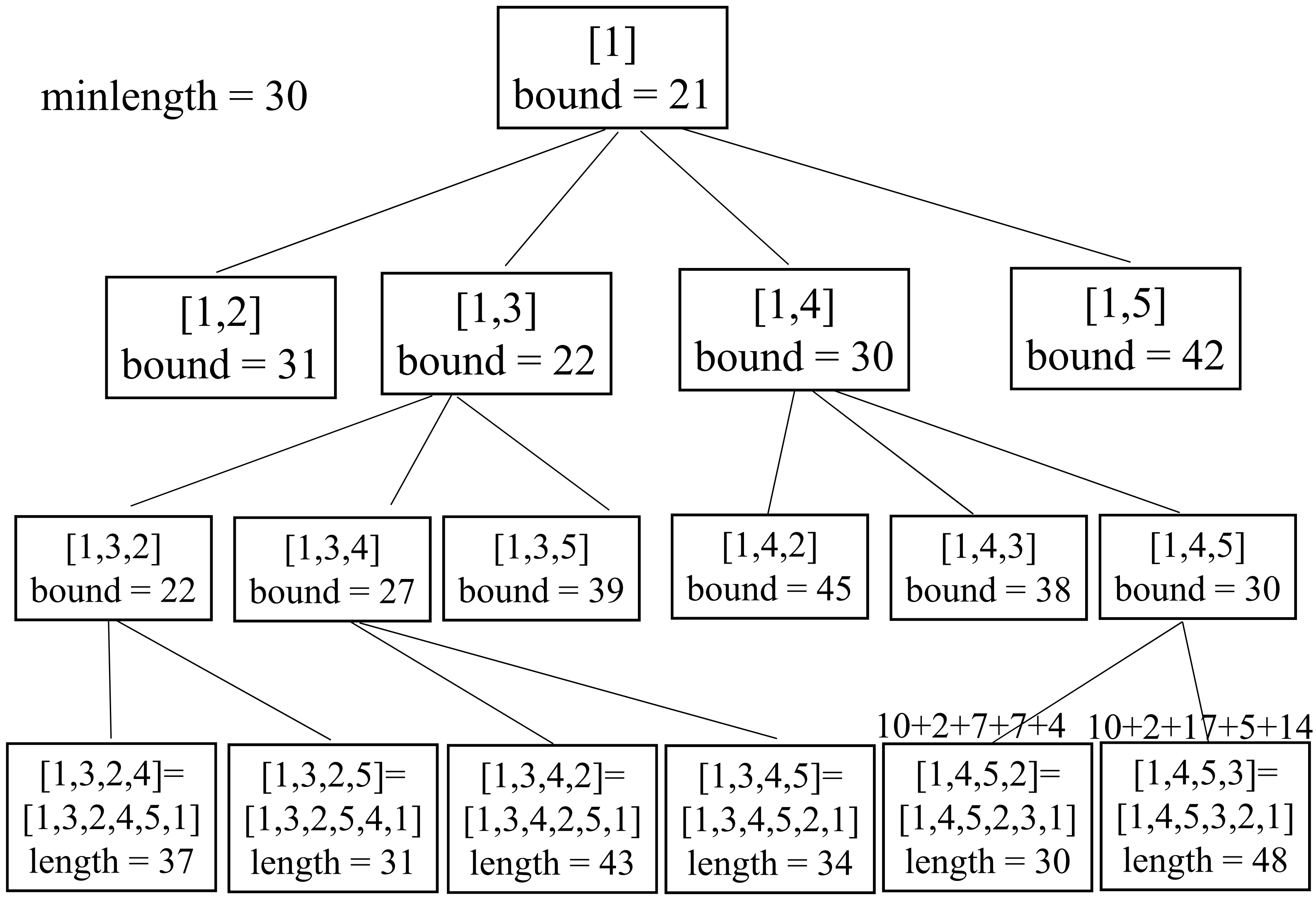

minlength = 31
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
4+7+7+7+18
4+7+2+7+14
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34


minlength = 31
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
10+7+7+4+17
[1,4,2]
bound = 45
non-promising
10+9+5+7+7
[1,4,3]
bound = 38
non-promising
10+2+7+4+7
[1,4,5]
bound = 30
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34


minlength = 30
[1]
bound = 21
[1,2]
bound = 31
non-promising
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
[1,4,2]
bound = 45
[1,4,3]
bound = 38
[1,4,5]
bound = 30
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34
10+2+7+7+4
[1,4,5,2]=
[1,4,5,2,3,1]
length = 30
10+2+17+5+14
[1,4,5,3]=
[1,4,5,3,2,1]
length = 48


minlength = 30
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
non-promising
[1,4,2]
bound = 45
[1,4,3]
bound = 38
[1,4,5]
bound = 30
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34
10+2+7+7+4
[1,4,5,2]=
[1,4,5,2,3,1]
length = 30
10+2+17+5+14
[1,4,5,3]=
[1,4,5,3,2,1]
length = 48


minlength = 30
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
non-promising
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
[1,4,2]
bound = 45
[1,4,3]
bound = 38
[1,4,5]
bound = 30
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34
10+2+7+7+4
[1,4,5,2]=
[1,4,5,2,3,1]
length = 30
10+2+17+5+14
[1,4,5,3]=
[1,4,5,3,2,1]
length = 48



minlength = 30
[1]
bound = 21
[1,2]
bound = 31
[1,3]
bound = 22
[1,4]
bound = 30
[1,5]
bound = 42
[1,3,2]
bound = 22
[1,3,4]
bound = 27
[1,3,5]
bound = 39
[1,4,2]
bound = 45
[1,4,3]
bound = 38
[1,4,5]
bound = 30
[1,3,2,4]=
[1,3,2,4,5,1]
length = 37
[1,3,2,5]=
[1,3,2,5,4,1]
length = 31
[1,3,4,2]=
[1,3,4,2,5,1]
length = 43
[1,3,4,5]=
[1,3,4,5,2,1]
length = 34
10+2+7+7+4
[1,4,5,2]=
[1,4,5,2,3,1]
length = 30
10+2+17+5+14
[1,4,5,3]=
[1,4,5,3,2,1]
length = 48

TSP with BestFS w/ Branch-and-bound pruning

- Each node is an object with fields:

  v.level – node's level in the tree

  v.path

  v.bound

<u>TSP with BestFS w/ Branch-and-bound pruning(n,W[ ][ ],opttour, minlength)</u>

```
PQ = ∅
r.level = 0
r.path = [1]
r.bound = bound(r)
minlength = ∞
insert(PQ,r)
while PQ ≠ ∅
    v = remove(PQ)
    if v.bound < minlength
        u.level = v.level+1
        for all i such that 2 ≤ i ≤ n and i is not in v.path
            u.path = v.path
            add i at the end of u.path
            if u.level == n-2    // check if next vertex completes the tour
                put index of only vertex not in u.path at the end of u.path
                put 1 at the end of u.path
                if length(u) < minlength
                    minlength = length(u)
                    opttour = u.path
            else
                u.bound = bound(u)
                if u.bound < minlength
                    insert(PQ,u)
```