Branch and Bound Algorithms

(Procédure de Séparation et Évaluation)

Frédéric Devernay

The knapsack problem

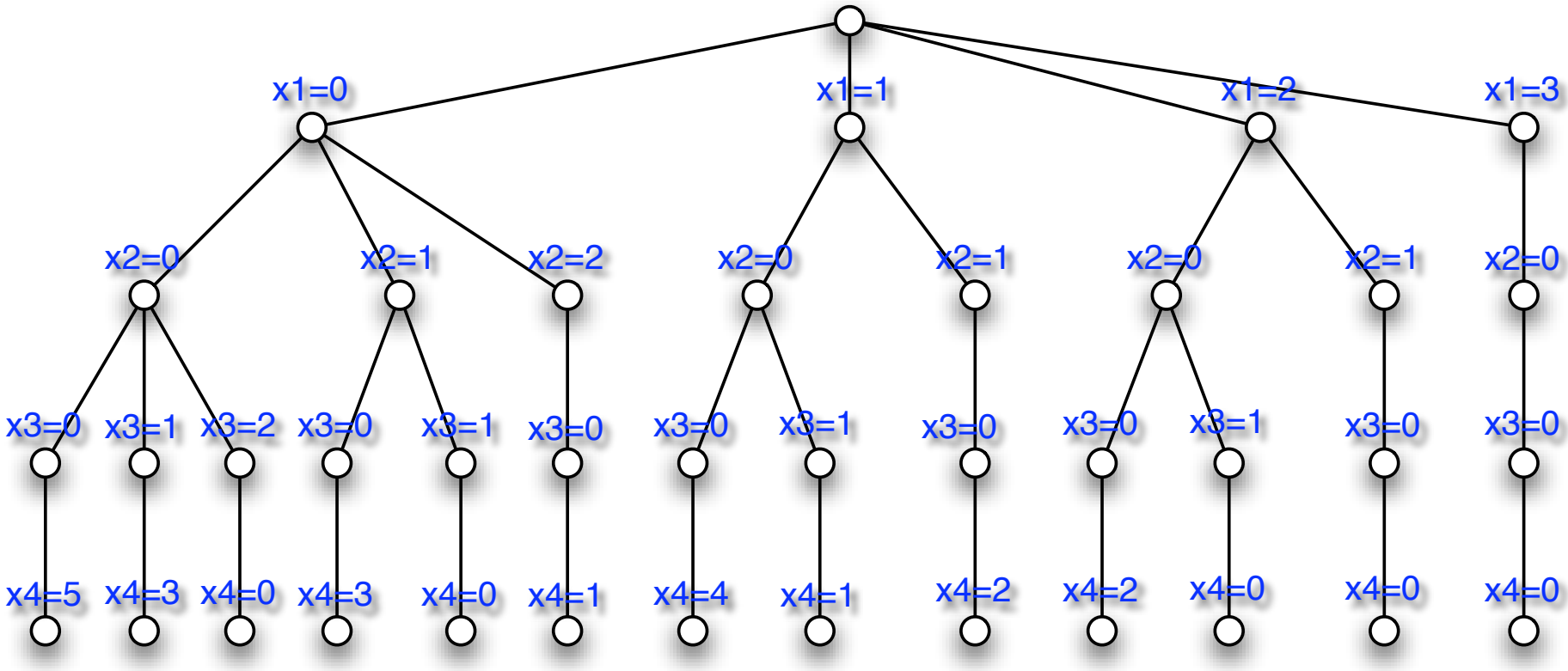
- Input
  - Capacity C
  - n items with weights v_i and values a_i
- Goal
  - Output a set of items S such that the sum of weights of items in S is at most C and the sum of values of items in S is maximized
- 0-1 knapsack : quantity of item i is 0 or 1

Solutions

- Construct a tree, where at each level i we have the different values for the number x_i of items of type i (at most C/v_i)
- Explore all the solutions in the graph (à la N-queens), keep the best... too costly!
- Apply greedy stategies
  - Highest Density First
  - Highest Value First
  - Lowest Weight First
  - Won't work : the solution may not fall into these categories!
- Dynamic programming
  - for special cases (e.g. the 0-1 knapsack problem, or when weights are integers)

Knapsack example

- $\max(4x_1+5x_2+6x_3+2x_4)$ $(a_i = \{4;5;6;2\})$
- $33x_1+49x_2+60x_3+32x_4 \leq 130$ $(v_i=\{33;49;60;32\}, C=130)$



Branch and Bound

- Branch and Bound is a general search method.
- Suppose we can easily evaluate upper- and lower- bounds procedures to the *minimization* problem (i.e. solve a *relaxed* problem)
- Starting by considering the root problem (the original problem with the complete feasible region), the lower-bounding and upper-bounding procedures are applied to the root problem.
- If the bounds match, then an optimal solution has been found and the procedure terminates.

Branch and Bound

- Otherwise, the feasible region is divided into two or more regions, these subproblems partition the feasible region.
- The algorithm is applied recursively to the subproblems. If an optimal solution is found to a subproblem, it is a feasible solution to the full problem, but not necessarily globally optimal.

Branch and Bound

- If the upper bound for a node is below the best known feasible solution (we're maximizing), no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration.
- The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the upper bounds on all unsolved subproblems.

Branch and Bound: summary

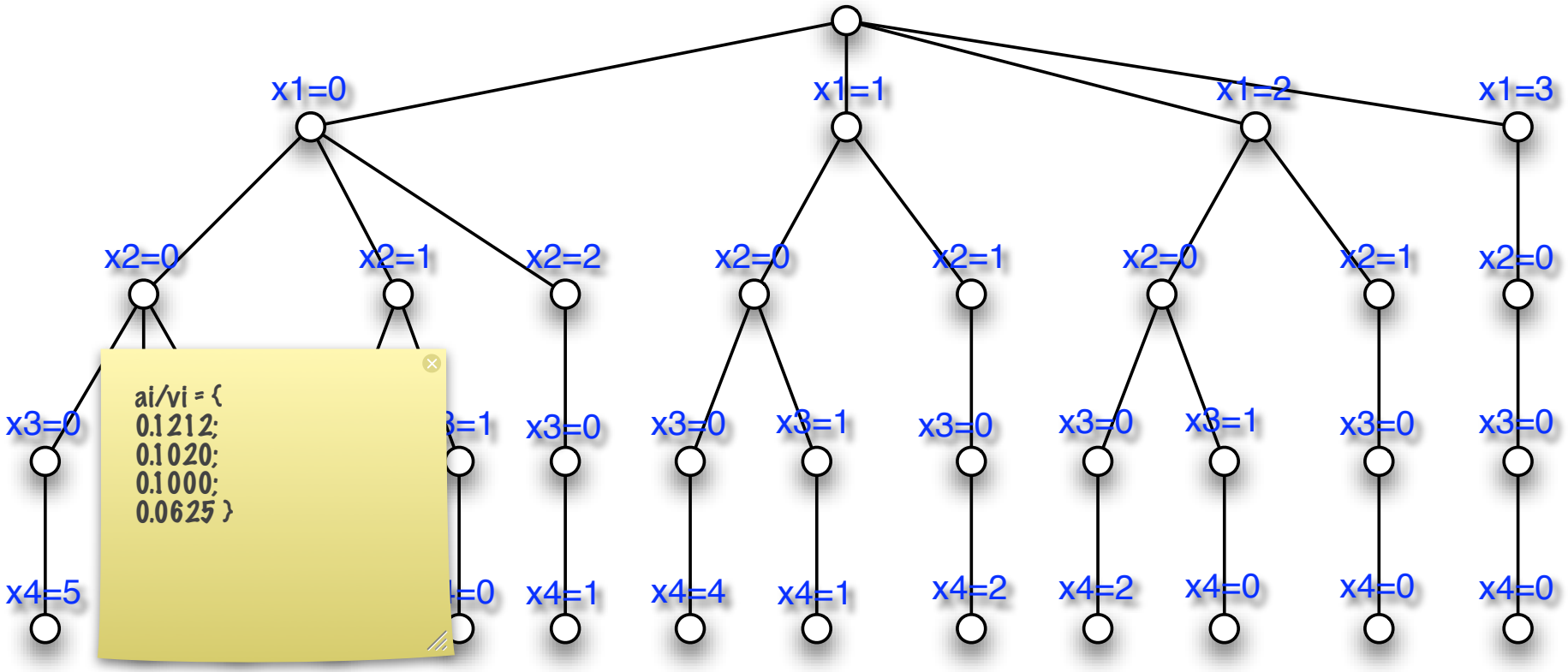
- Initialization: best_so_far=-∞
- Iteration: Usually select most recent subproblem or subproblem with best bound
  1. Branch: divide to create two (or more) subproblems
  2. Bound: obtain bound by solving the *relaxation* of each subproblem (lower and upper bounds)
  3. Fathom: eliminate subproblem if
     - Relaxed solution is exact
     - Upper bound < best_so_far (we're maximizing)
     - Relaxation infeasible
- Stop when no more subproblems

Branch and Bound

- Choice of the active child node :
  - depth-first
  - breadth-first (bad)
  - best evaluation (total weight) first
  - mixed : depth-first, followed by best evaluation during backtracking
- If a node only has one child, evaluate the child itself

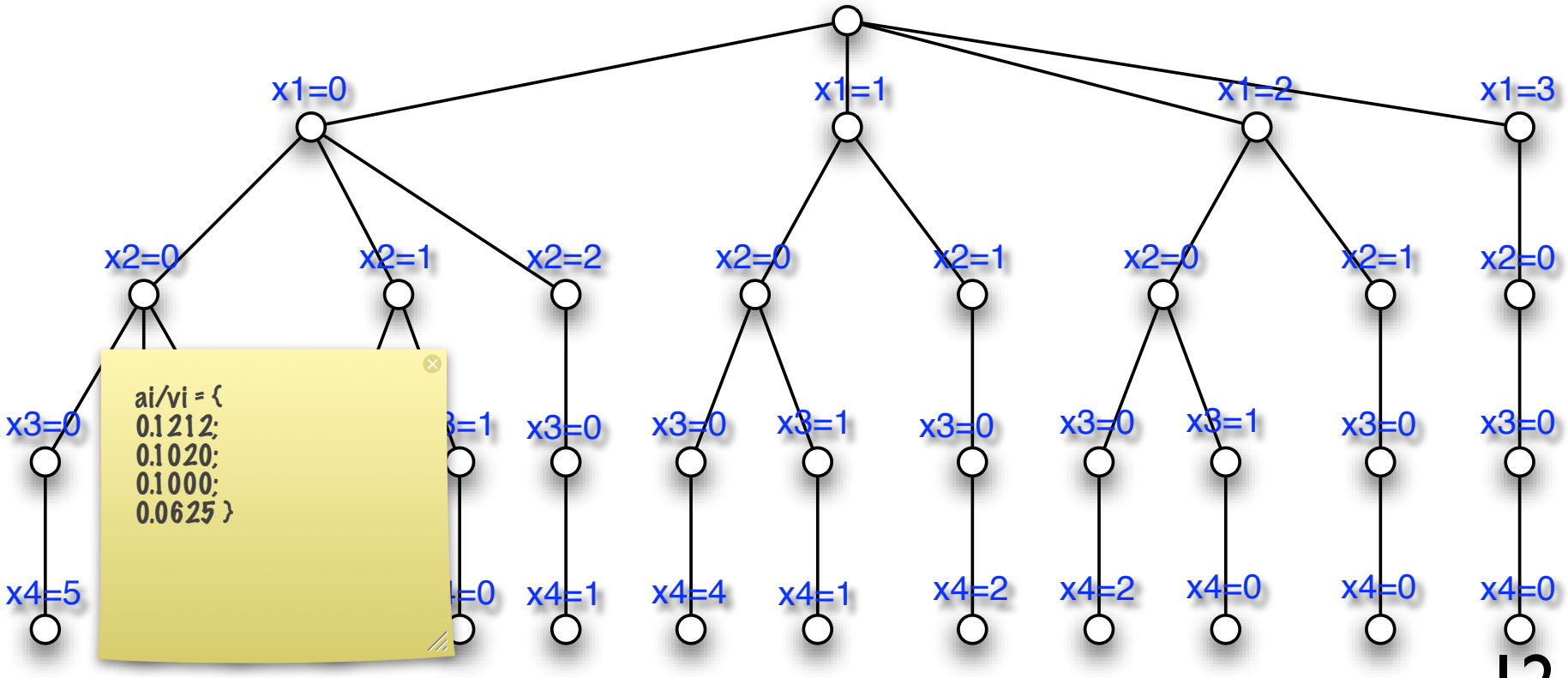
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



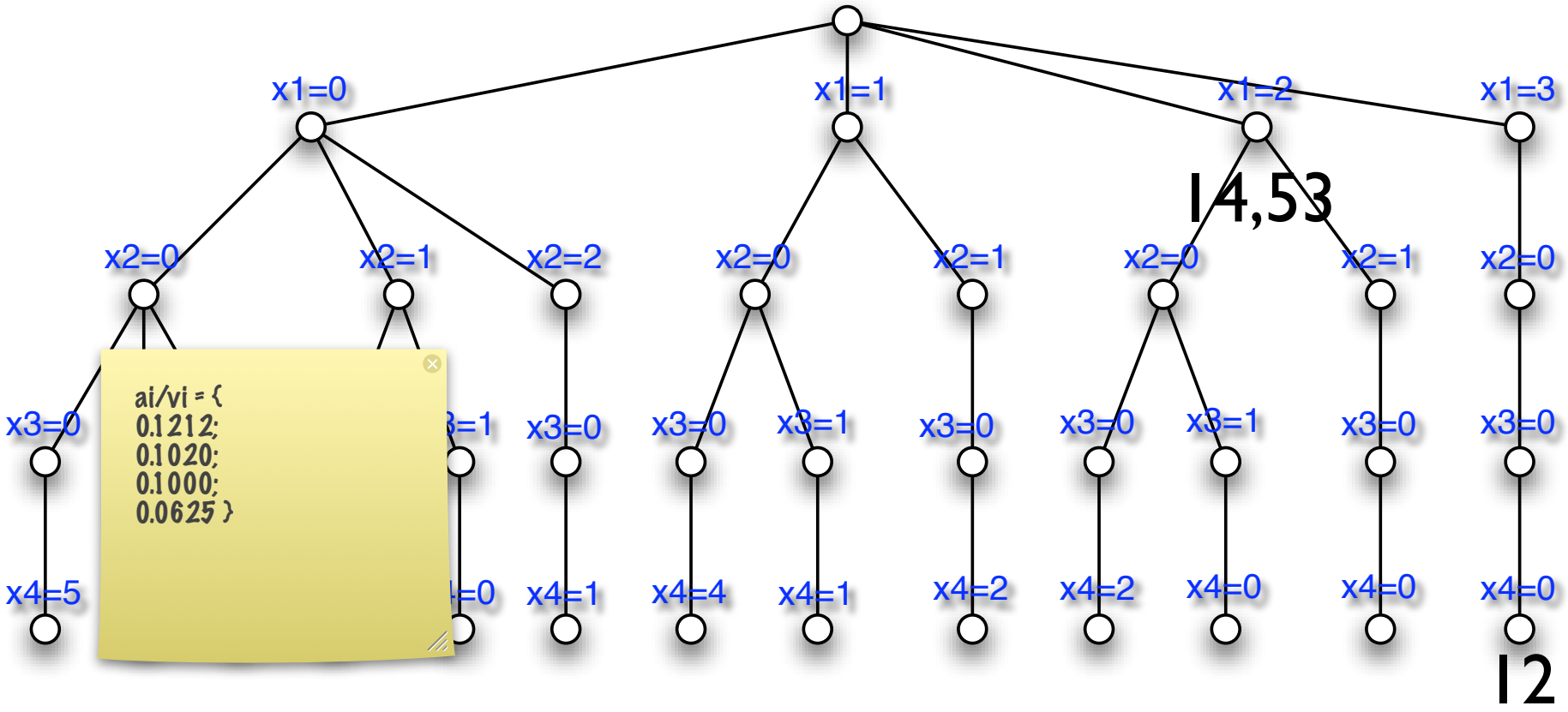
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



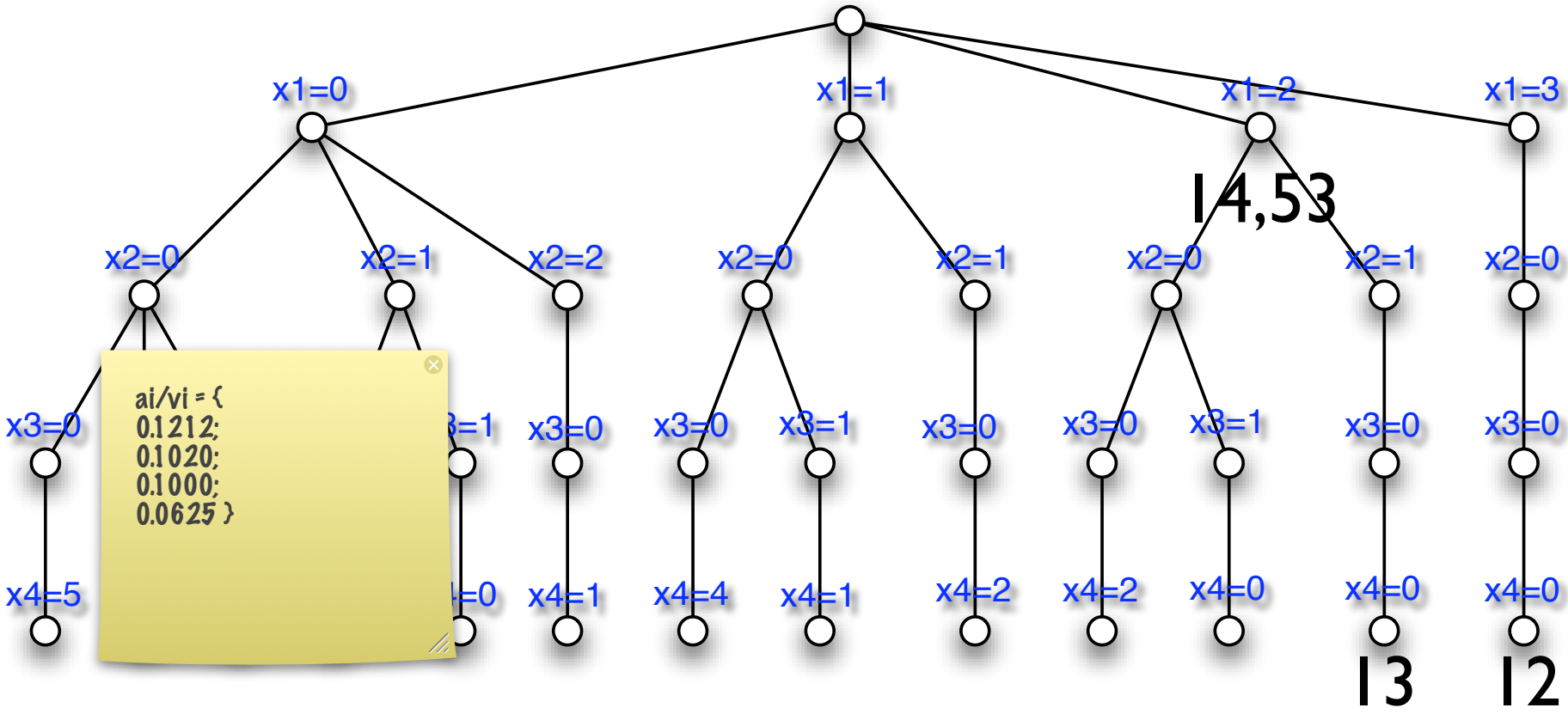
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



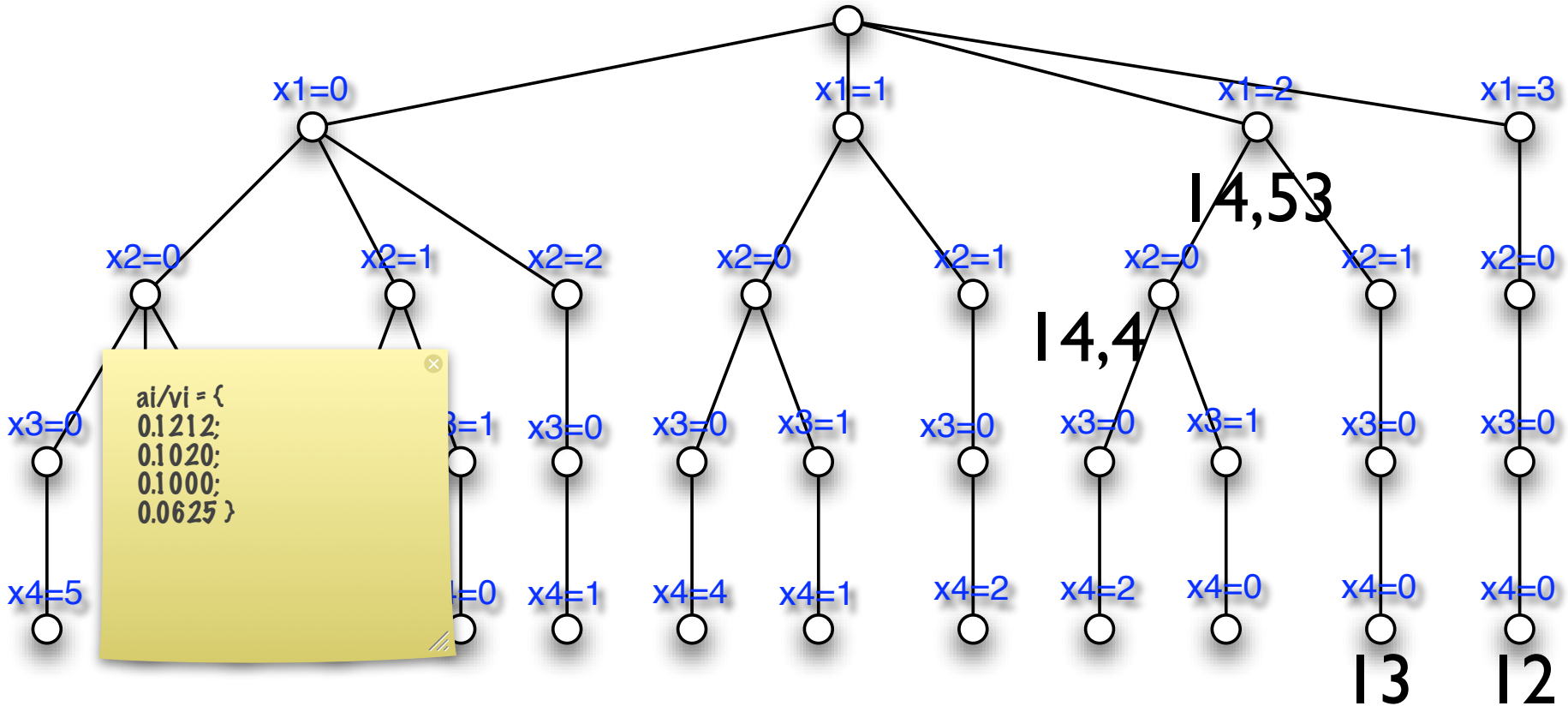
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



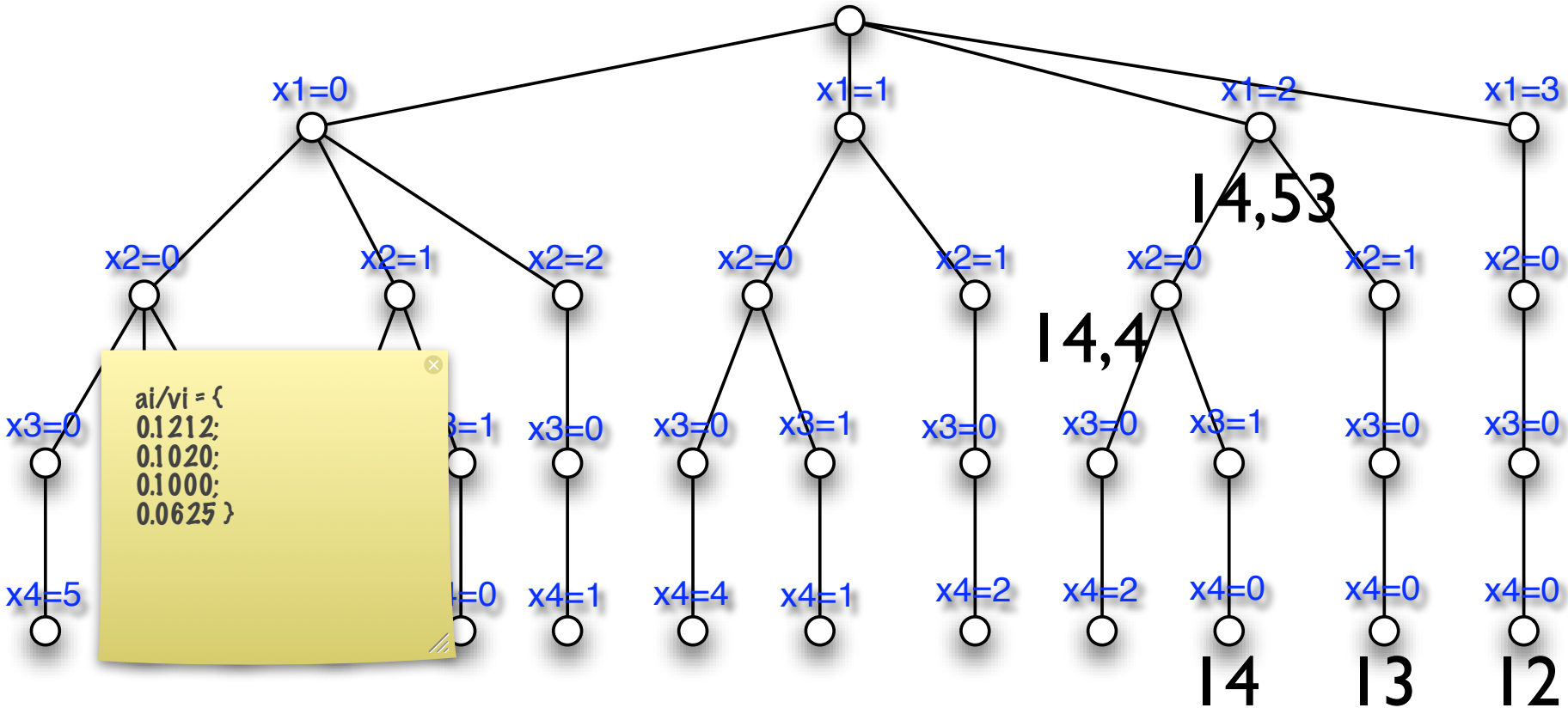
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



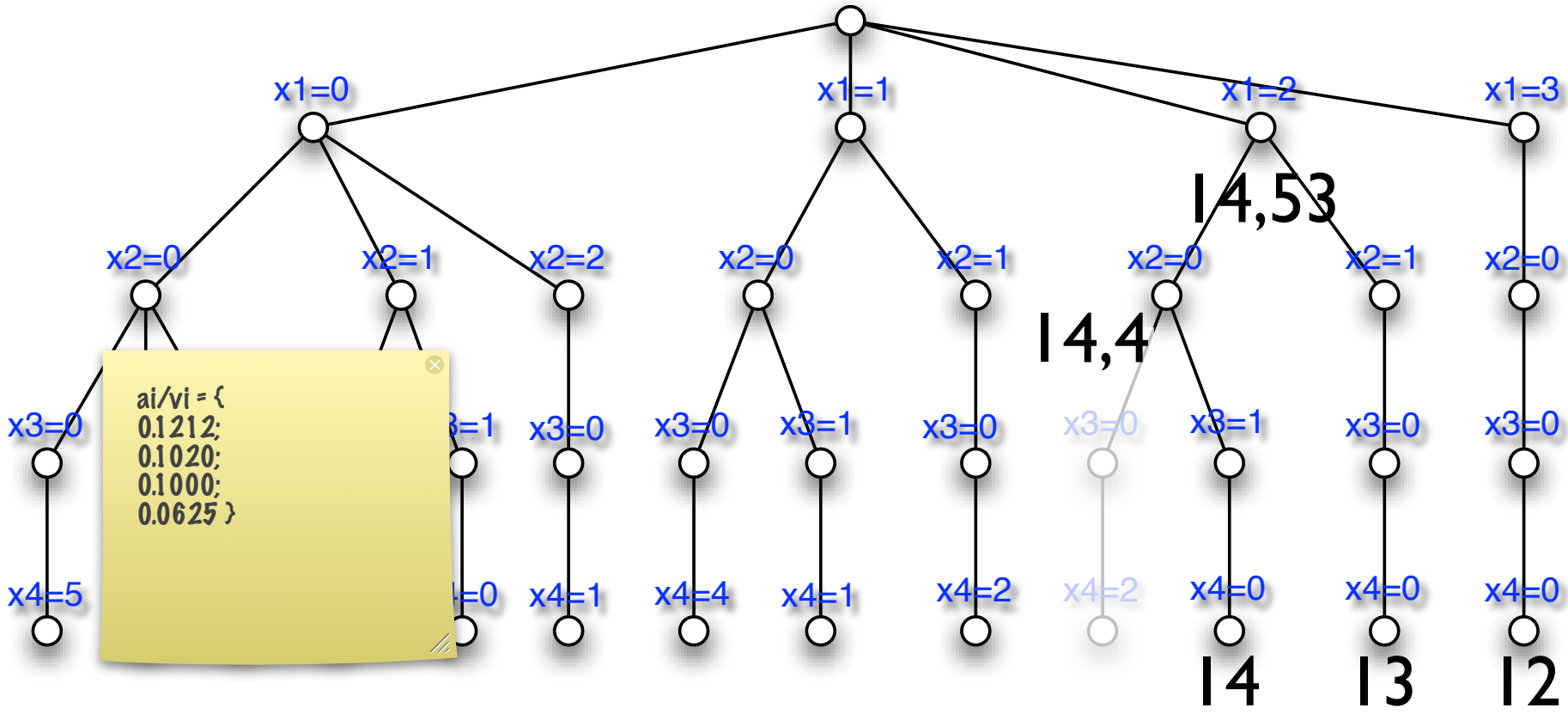
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



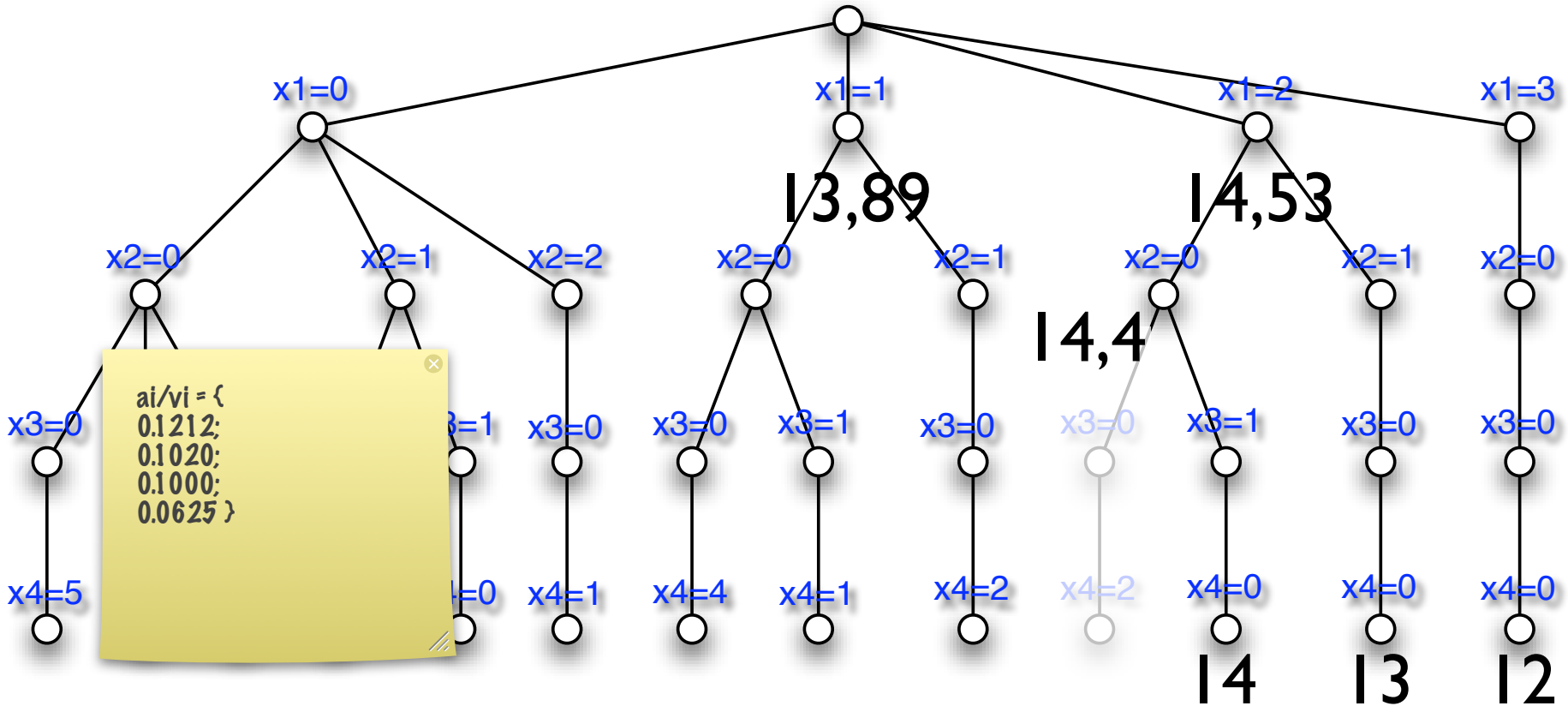
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



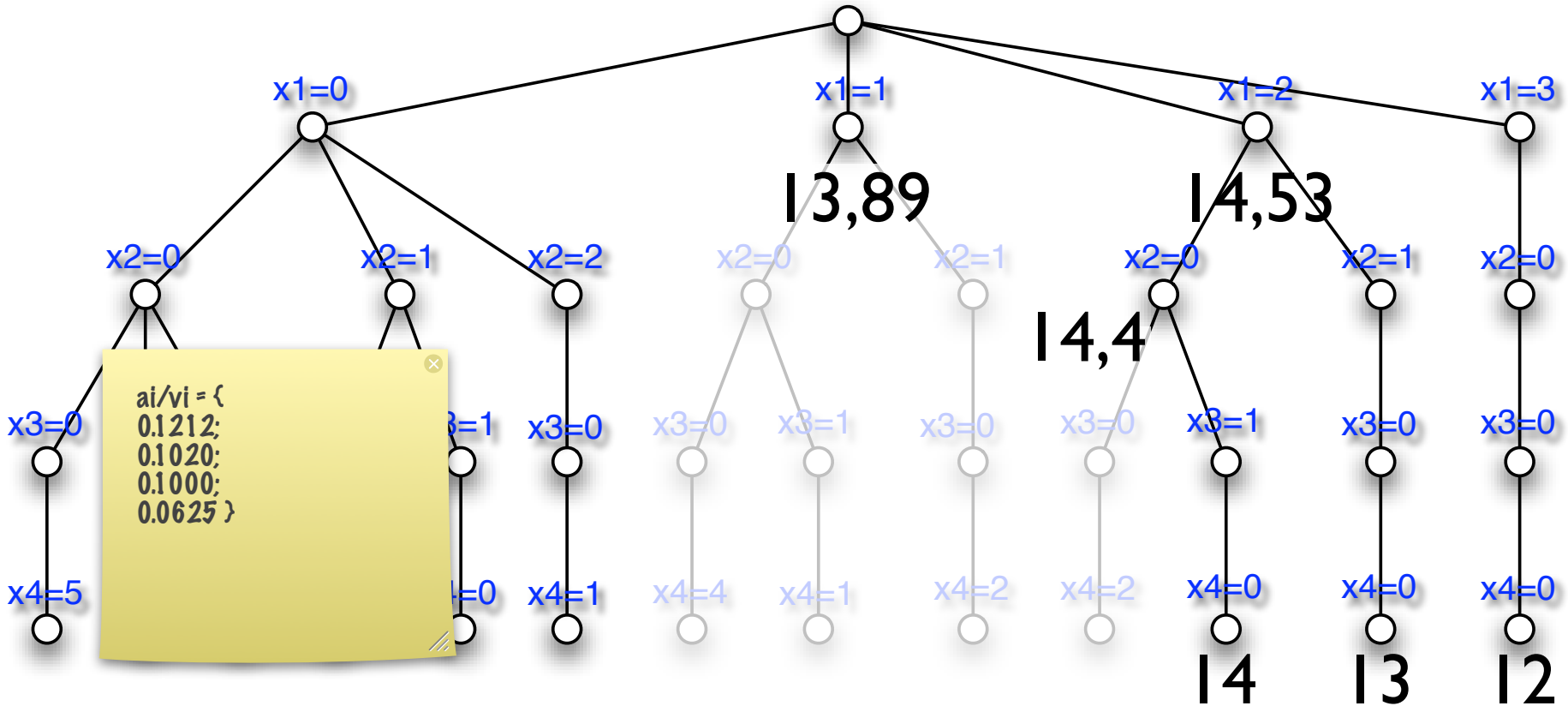
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



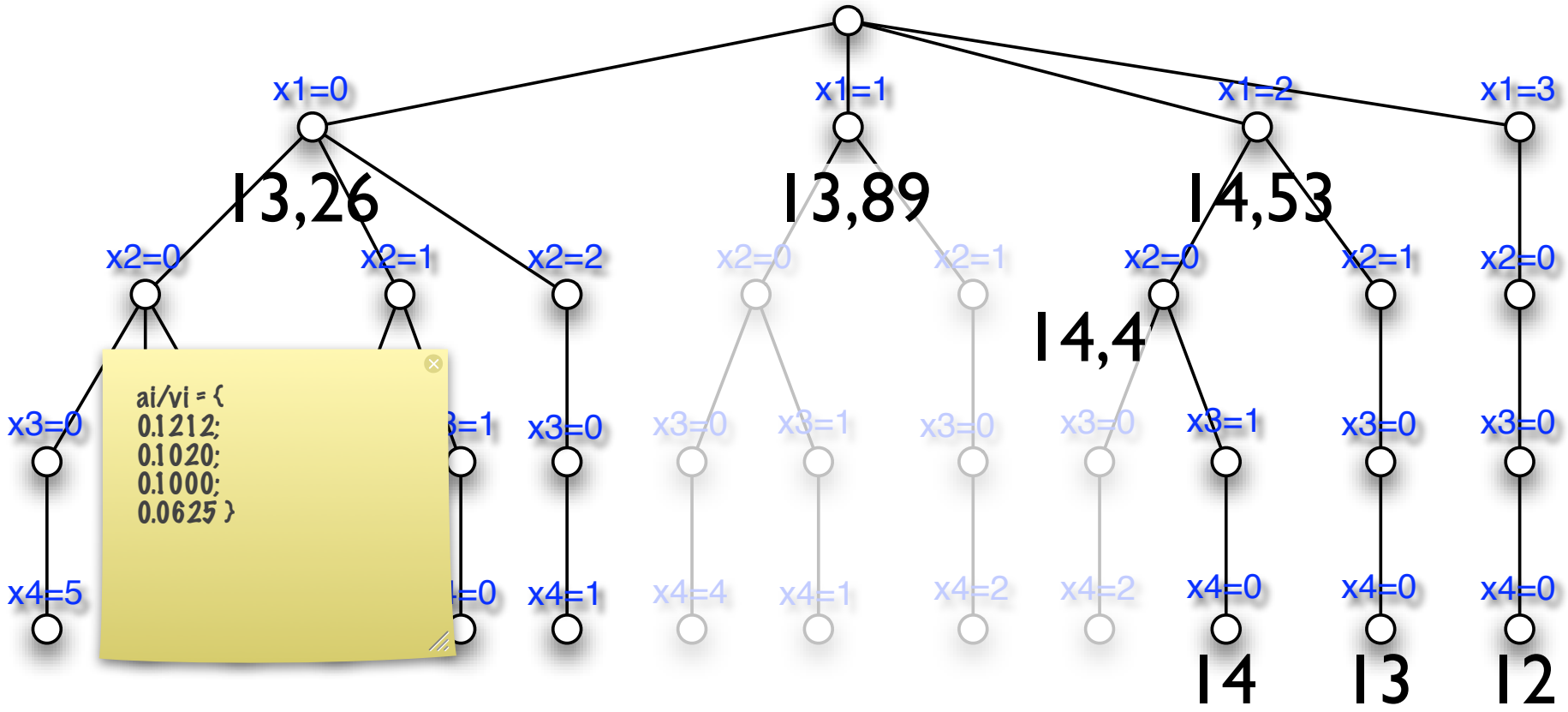
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



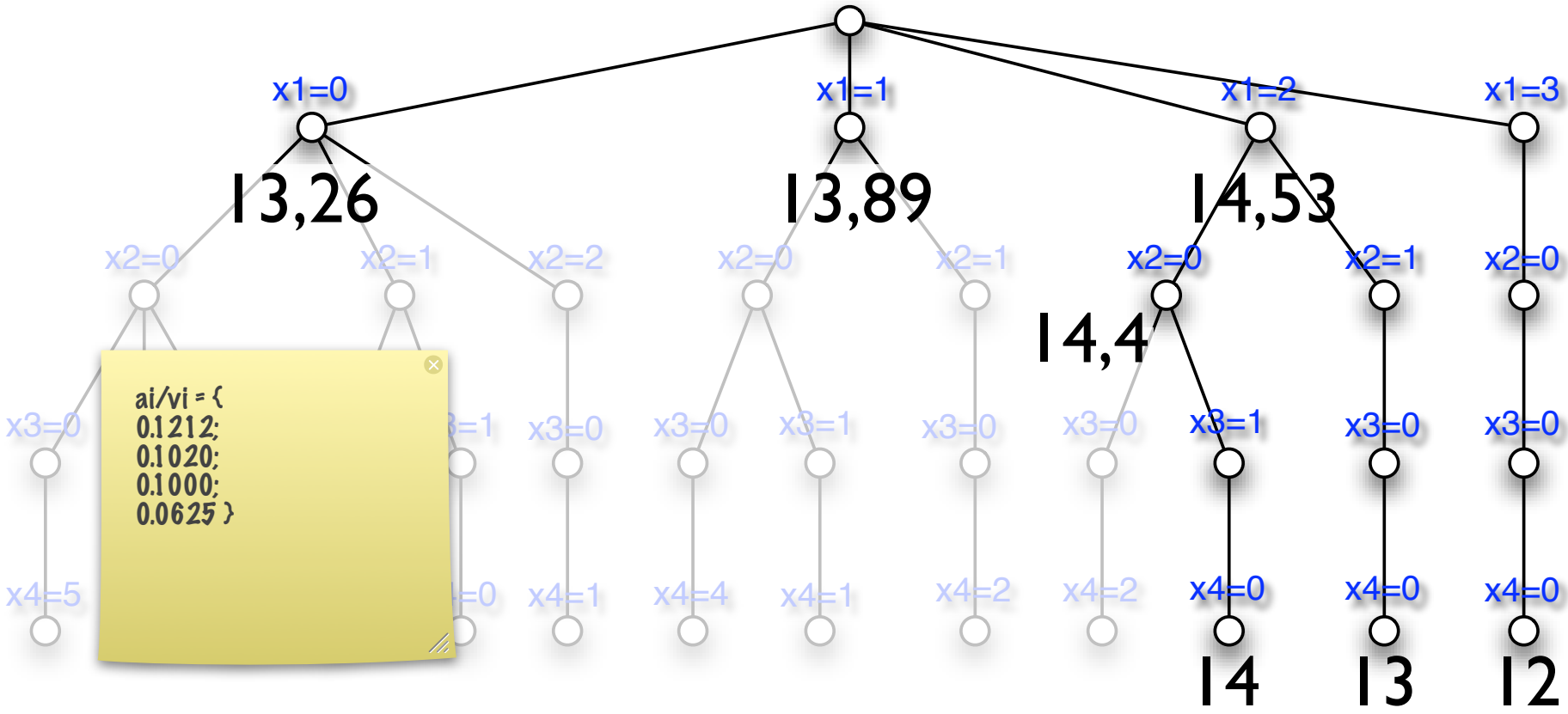
Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



Knapsack example

- $\max(4.x_1+5.x_2+6.x_3+2.x_4)$ ($a_i=\{4;5;6;2\}$, $v_i=\{33;49;60;32\}$, $C=130$)
- $33x_1+49x_2+60x_3+32x_4 \leq 130$, x_i's are sorted by a_i/v_i ratio
- upper bound: fill with the item with the highest a_i/v_i ratio
- a_i's are integer, so a better solution has to be $\geq$ best_so_far+1



Knapsack problem by dynamic programming (integer v_i only)

```
for j:=1 to N do            {Go through each item}
  for i := 1 to M do begin  {Consider each size knapsack}
    if i >= size[j] then
      if (cost[i] < cost[i-size[j]] + value[j]) then begin
        cost[i] := cost[i-size[j]] + value[j];
        best[i] := j
      end;
```

- consider all knapsack sizes from 1 to M
- cost[i] is the highest value that can be achieved with a knapsack of capacity i and is initialised to zero;
- best[i] is the last item that was added to achieve that maximum. First we calculate the best we can do only using objects of type 1 (j=1). Then we calculate the best considering items of type 1 and 2 (using our result for just type 1). And so on.

knapsack problem example

```
Item   Size   Value
1      3      6
2      2      5          bag size: 4
3      1      2
```

- item 1:
  - cost[3]:=6; best[3]:=1
  - cost[4]:=6; best[4]:=1
- item 2:
  - cost[2]:=5; best[2]:=2
  - cost[4]:=10; best[4]:=2
- item 3:
  - cost[1]:=2; best[1]:=3
  - cost[3]:=7; best[3]:=3
- end situation: cost = {2,5,7,10}, best = {3,2,3,2}
- backtracking gives best[4]=2, best[4-size[2]]=2