Backtracking and Branch-and-Bound Strategies

- State Space Trees
  - problem states, decision sets
  - fixed-tuples vs. variable-tuples
  - bounding functions, search strategies
  - examples: sum of subsets, 0/1 knapsack
- Backtracking
  - DFS with bounding
- Branch-and-Bound
  - FIFO, LIFO, and LC branch-and-bound searches

Backtracking and Branch-and-Bound Strategies

Many problems require making a sequence of decisions that satisfy certain constraints.

- The 0/1 knapsack problem: making *n* decisions regarding whether to include each of *n* objects without exceeding the sack's capacity
- The graph coloring problem: making *n* decisions on choosing a color (out of *k* colors) for each of the *n* vertices without using the same color for the two end vertices of an edge

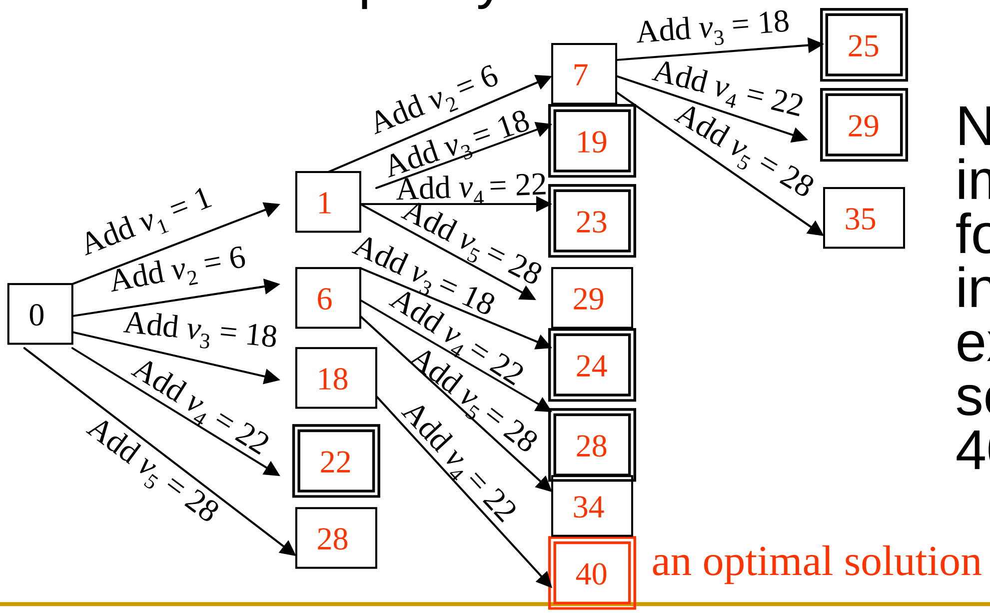
Let $x_1, x_2, \ldots, x_k$, denote *k* decisions made in solving a problem, $1 \le k \le n$, where each $x_i \in S_i$, and *n* is the maximum number of decisions to be made. Let P_k denote the set of all these *k*–tuples $(x_1, x_2, \ldots, x_k)$. Each such tuple is called a *problem state*; a *goal state* is one that corresponds to a final solution.

Given a problem state $(x_1, x_2, \ldots, x_{k-1})$, the *decision set* $D_k(x_1, x_2, \ldots, x_{k-1}) = \{x_k \in S_k \mid (x_1, x_2, \ldots, x_k) \in P_k\}$, i.e., all possible decisions in stage *k* having made *k* –1 previous decisions $x_1, x_2, \ldots, x_{k-1}$.

State Space Trees

- The collection of $D_k(x_1, x_2, \ldots, x_{k-1})$, $1 \le k \le n$, form a tree in which the root corresponds to the initial state (an empty set), the child nodes of the root correspond to the set $D_1 = P_1$. For each of problem states (x_1) in P_1, its child nodes include those in $D_2(x_1)$,...

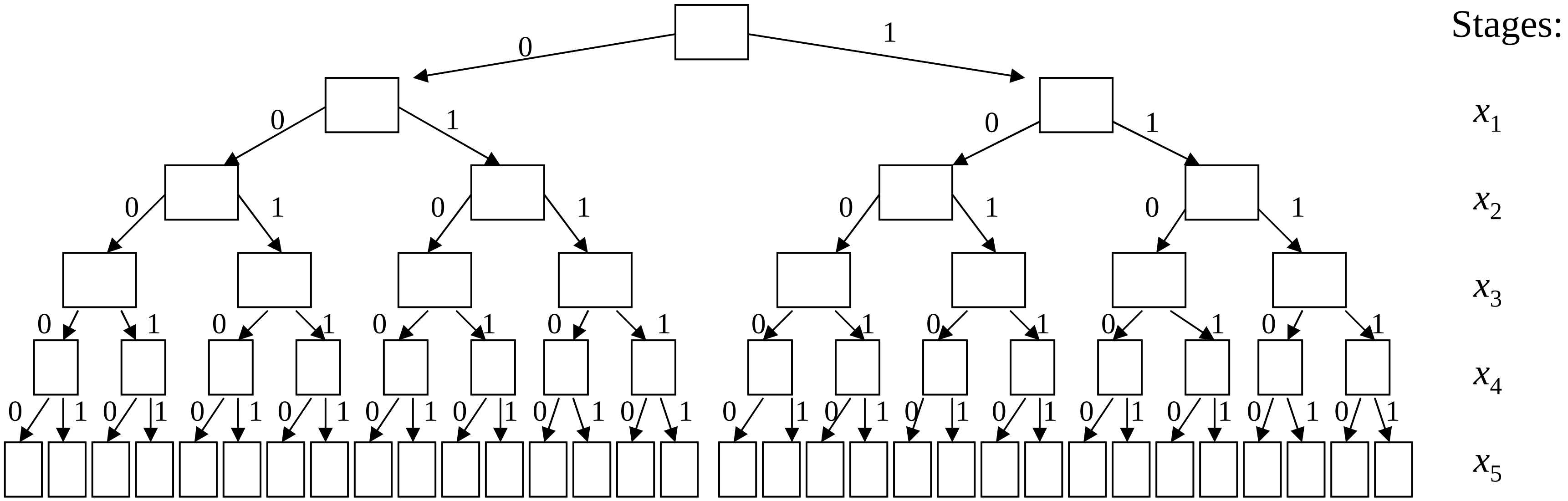
Example: The 0-1 knapsack problem of 5 objects with associated weights $w[1..5] = \{1, 2, 5, 6, 7\}$, values $v[1..5] = \{1, 6, 18, 22, 28\}$, and sack's capacity $W = 11$.



Note that the constraint imposed at each stage is for the total weight of included objects not exceeding 11. The optimal solution with a total value 40 is highlighted.

State Space Trees

- Another way to present the state space tree uses *fixed-tuples*, in which the goal states are of the same length. The fixed-tuple state space for the same knapsack problem:

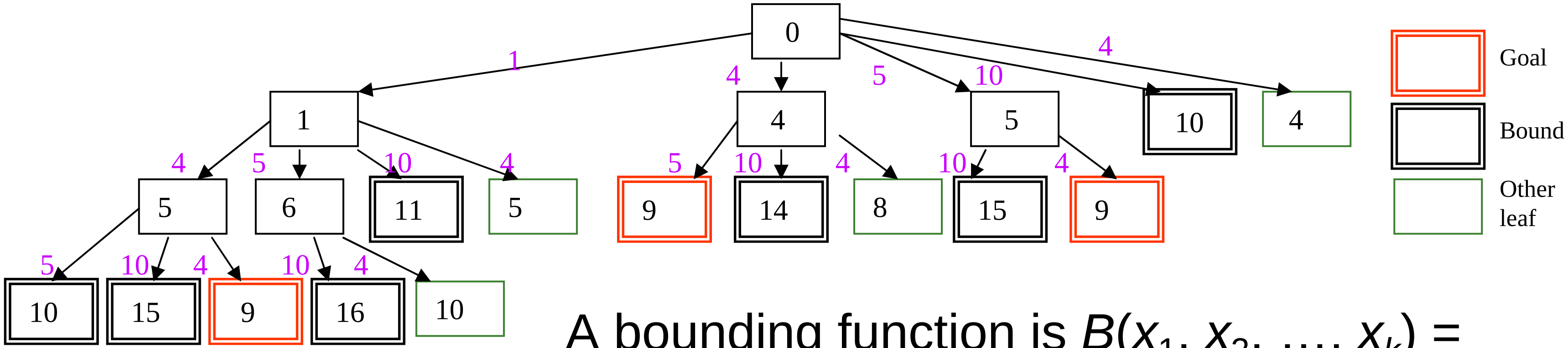


Note that each decision $x_i = 0$ or 1, for $1 \le i \le 5$, depending on whether object *i* is excluded or included.

Searching the State Space Trees

- Solutions can be found via a systematic search of the tree.
  - If no descendants of a node X can lead to a goal state, then node X is *bounded*, and the subtree rooted at X *is skipped (pruned)*.
  - A good *bounding function* can improve the search algorithm's efficiency.

Example. The sum-of-subsets problem: Find a sublist of the list $A[1..5] = \{1, 4, 5, 10, 4\}$ with the sum of 9. (Answer: choose 4 and 5.)



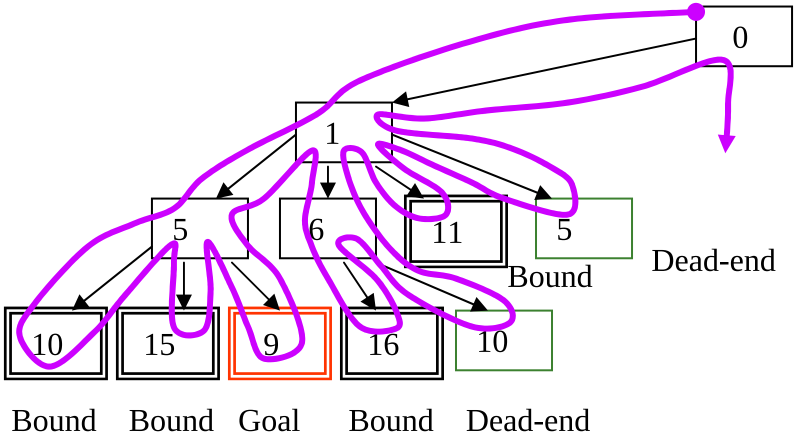
A bounding function is $B(x_1, x_2, \ldots, x_k)$ = true if $\sum_{i=1}^{k} A[x_i] \geq sum$; false otherwise.

Backtracking

- A general-purpose design strategy based on searching the state space tree associated with a given problem.
- Apply depth-first search of the state space tree starting from its root, maintaining necessary information about the current state and using a bounding function to prune the search space (reached a goal state or no need to search further).

Example. A portion of the state space tree for the sum-of-subsets problem of the preceding page, and the backtracking path:

$A[1..5] = \{1, 4, 5, 10, 4\}$, $sum = 9$

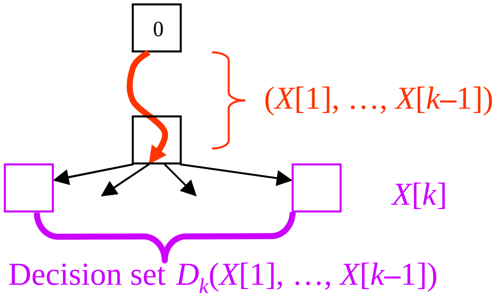


Backtracking

- Procedure Sum-of-Subsets-Recursive (k)

```
// $X[0..n]$ a global array, in which ($X[1], …, X[k]$) is the current
// state, $X[i]$ is the index of the $i$th selection, and $X[0] = 0$
// $A[1..n]$ a global array that contains the values of the list
// Call this procedure with $k = 0$ to start the search process
// The bounding condition is when the sum of the values selected
// in the current state is $\geq sum$
$k$++
for child = $X[k-1] + 1$ to $n$ do  // try every child of current node
    $X[k]$ = child
    if $A[X[1]] + … + A[X[k]] < sum$ then
    // current state not bounded, search deeper
                call Sum-of-Subsets-Recursive ($k$)
    // else, the current state is bounded, prune the subtree
    else if $A[X[1]] + … + A[X[k]] = sum$ then
                output-goal-state ($X[1], …, X[k]$)
end-for-loop
```

A General Backtracking Procedure



```
procedure Backtrack-recursive($k$)
  // $X[0..n]$ a global array, in which
  // $(X[1], …, X[k])$ is the current state;
  // $D_k$ is the decision set for current states;
  // The output consists of all goal states
  // that are descendants of the
  // current state $(X[1], …, X[k])$;
  // Call the procedure with $k = 0$ to start
  // the search
  $k$++
  for each decision $x \in D_k (X[1], …, X[k–1])$do
    $X[k] = x$
    if not Bounded($X[1], …, X[k]$) then
       // search deeper
       Backtrack-recursive($k$)
    // otherwise, prune the search tree
    else if $(X[1], …, X[k])$ is goal state then
       output-goal-state($X[1], …, X[k]$)
  end-for-loop
```

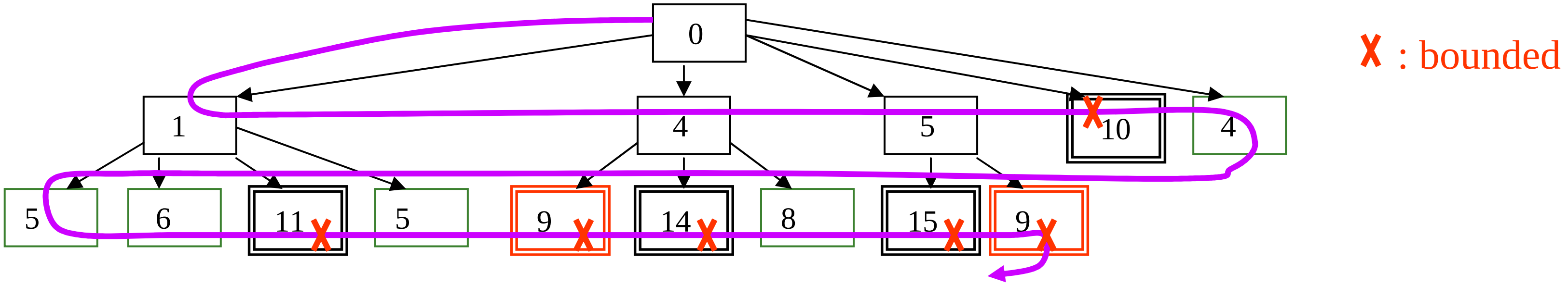
```
procedure Backtrack // non-recursive
  // $X[0..n]$ a global array, in which $(X[1], …, X[k])$ is the
  // current state; $D_k$ is the decision set for current state;
  // The output consists of all goal states
  $k = 1$
  while $k \geq 1$do    // repeat until returning to the root ($k$=0)
    while there is another un-tried node $x$ in $D_k$ do
      delete $x$ from the decision set $D_k (X[1], …, X[k–1])$
      $X[k] = x$
      if not Bounded($X[1], …, X[k]$) then
         exit while loop
      // otherwise, prune the search tree
      else if $(X[1], …, X[k])$ is goal state then
         output-goal-state($X[1], …, X[k]$)
    end-while-loop
    if $x$ = NULL then  // exhausted all decisions in $D_k$
      $k$--  // backtrack to previous level
    else $k$++ // move to next level
  end-while-loop
```

Note that backtracking traverses an *implicit* search tree; its worst-case time complexity is O(tree size) and space complexity O(n), n = depth of state tree

Branch-and-Bound (FIFO, LIFO, or LC)

- When a node is visited the first time (called an *E-node*), all its children are generated and saved into a data structure (called *live-nodes*) if the child is not bounded; the structure could be a queue (FIFO), a stack (LIFO), or a priority queue (LC, or Least-Cost). Exactly one node is pulled out the live-node list at a time, which becomes an E-node, to be "expanded" next.

Example. (A portion of) the FIFO branch-and-bound path:



FIFO branch-and-bound does a breadth-first search (BFS): nodes are expanded from top down, left to right at each level, dropping those that are bounded.

A General Branch-and-Bound Procedure

```
procedure Branch-and-bound
    // $X[0..n]$ a global array, in which ($X[1]$, …, $X[k]$) is the
    // current state; $D_k$ is the decision set for current state;
    // The output consists of all goal states
    call Allocate-Node(root-node)  // create a root node
    root-node.parent =  NULL; $k = 0$
    live-nodes = {root-node}  // initialize live-node list
    while live-nodes $\neq \emptyset$ do
        E-node = select-next-node(live-nodes, $k$)
        for each $X[k] \in D_k$ (E-node) do
            call Allocate-Node(child-node)
            child-node.info = $X[k]$
            child-node.parent = E-node
            if not bounded(child-node) then
                call add-live-nodes(child-node)
            //otherwise, prune the tree
            if goal(child-node) then  // a goal state is reached
                create the state ($X[1]$, …, $X[k]$) by following the
                parent links from child-node to the root;
                output-goal-state($X[1]$, …, $X[k]$)
        end-for-loop
    end-while-loop
```

A General Branch-and-Bound Procedure

- When a stack is used for storing the live nodes, LIFO branch-and-bound essentially performs a DFS but exploring first the rightmost child node at each level.
- FIFO branch-and-bound may be more efficient than LIFO BB when a goal state exists at a shallow level of the tree; otherwise, FIFO BB may be expensive
- The LC (least-cost) branch-and-bound strategy uses heuristics to determine the best node for expansion from the current live nodes.