Backtracking and Branch-and-Bound

Textbook Reading:

Chapter 9

- Section 9.1 State Space Trees
- Section 9.2 Backtracking
- Section 9.3 Branch-and-Bound

Backtracking vs. Branch-and-Bound

- Backtracking and branch-and-bound design strategies are applicable to problems whose solutions can be expressed as sequences of decisions.
- Both are based on a search of an associated **state space tree** modeling all possible sequences of decisions, but differ in the way the state space tree is searched.
- Backtracking involves a depth-first search of the state space tree and unlike branch-and-bound searches doesn’t require explicitly maintaining the tree.

Bounded Nodes

Both involve searching a state space tree utilizing a **bounding function** to reduce the number of nodes that need to be looked at.

State Space Tree

- The decision x_k at stage k must be drawn from a finite set of choices. For each $k > 1$, the choices available for decision x_k may be limited by the choices that have already been made for $x_1,\ldots,x_{k-1}$.
- Let n denote the maximum number of decision stages that can occur.
- Let P_k denote the set of all possible sequences of k decisions, represented by k-tuples $(x_1,x_2,\ldots,x_k)$. Elements of P_k are called **problem states**, and problem states that correspond to solutions to the problem are called **goal** states.
- Given a problem state $(x_1,\ldots,x_{k-1}) \in P_{k-1}$, let $D_k(x_1,\ldots,x_{k-1})$ denote the **decision set** consisting of the set of all possible choices for decision x_k. Let $\varnothing$ denote the null tuple ( ). Note that $D_1(\varnothing)$ is the set of choices for x_1.
- The decision sets $D_k(x_1,\ldots,x_{k-1})$, $k = 1,\ldots,n$, determine a decision tree T of depth n, called the **state space tree**.
- The nodes of T at level k, $0 \le k \le n$, are the problem states $(x_1,\ldots,x_k) \in P_k$ (P_0 consists of the null tuple). For $1 \le k < n$, the children of $(x_1,\ldots,x_{k-1})$ are the problem states $\{(x_1,\ldots,x_k) \mid x_k \in D_k(x_1,\ldots,x_{k-1})\}$.

Sum of Subsets

Sum of Subsets problem. Given a multiset $A = \{a_0,...,a_{n-1}\}$ of n positive integers, together with a positive integer *Sum*, find a subset whose elements sum to *Sum*.

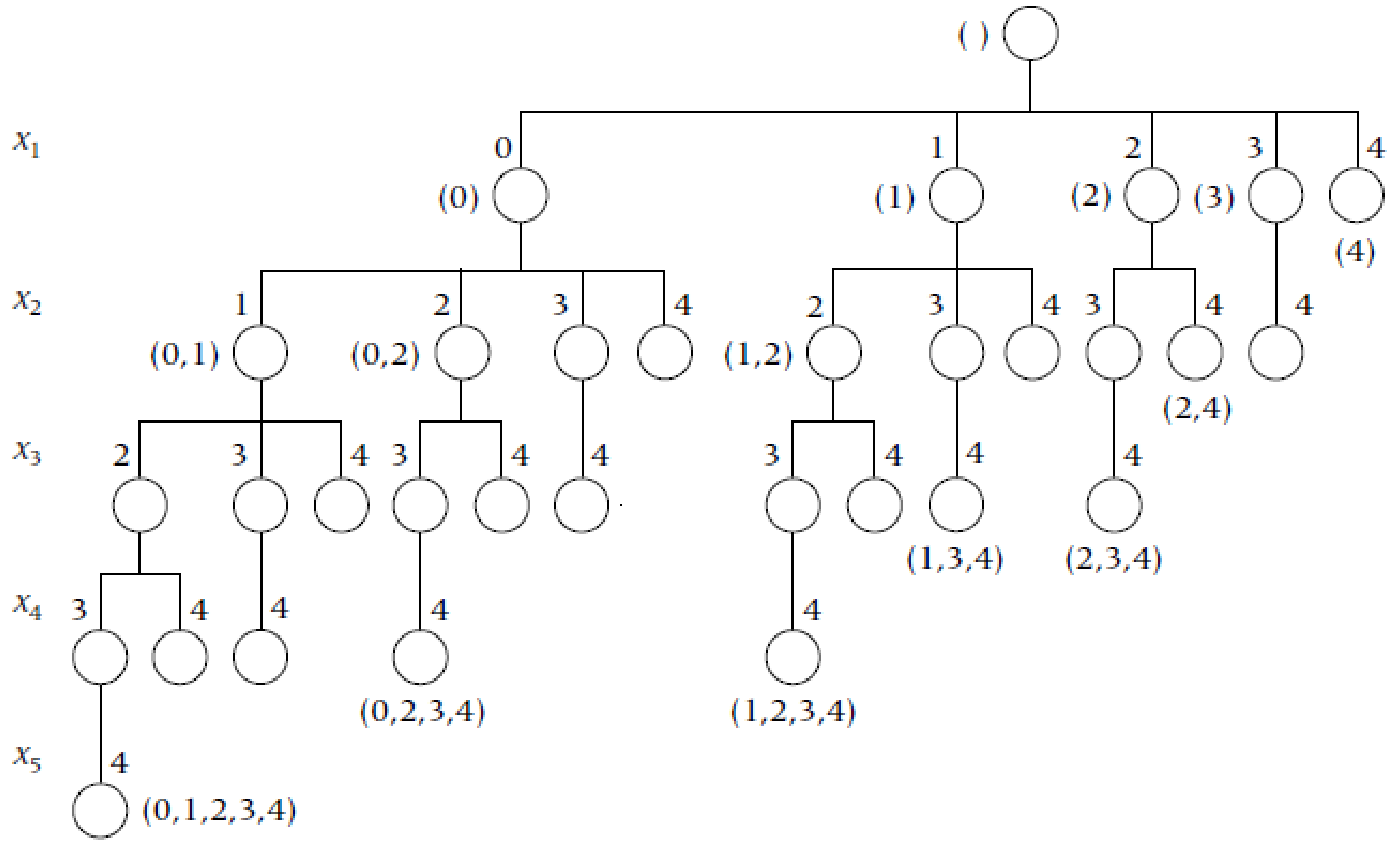
- The sum of subsets problem can be interpreted as the problem of making correct change, where a_i represents the denomination of the ith coin, $i = 0,...,n - 1$, and *Sum* represents the desired change.
- This differs from the version of the coin-changing problem since now a limited number of coins of each denomination are available.
- For example, consider the multiset $A = \{25,25,1,1,5,10,10,10,25\}$. The denominations are 1, 5, 10, 25, which occur with multiplicities 2, 1, 3, 3, respectively.
- The Sum of Subsets is a classical NP-complete problem, and there is no known worst-case polynomial time algorithm for determining whether a solution exists.

Variable-Tuple Model for Sum of Subsets

- The decision sequence can be represented by the *k*-tuple $(x_1, \dots, x_k) = (i_1, \dots, i_k)$, where x_j corresponds to the decision to choose element a_{i_j} at stage $j, 1 \leq j \leq k$.
- Suppose problem state $(x_1, \dots, x_{k-1})$ has occurred.
- Then the decision has been made to choose elements $a_{x_1}, a_{x_2}, \dots, a_{x_{k-1}}$.
- The available choices for decision x_k are $a_{x_{k-1}+1}, a_{x_{k-1}+2}, \dots, a_n$, yielding

$$D_{k(x_1, \dots, x_{k-1})} = \{x_{k-1} + 1, x_{k-1} + 2, \dots, n\}$$

Fixed-tuple state space tree T for the sum of subsets problem with $n = 5$



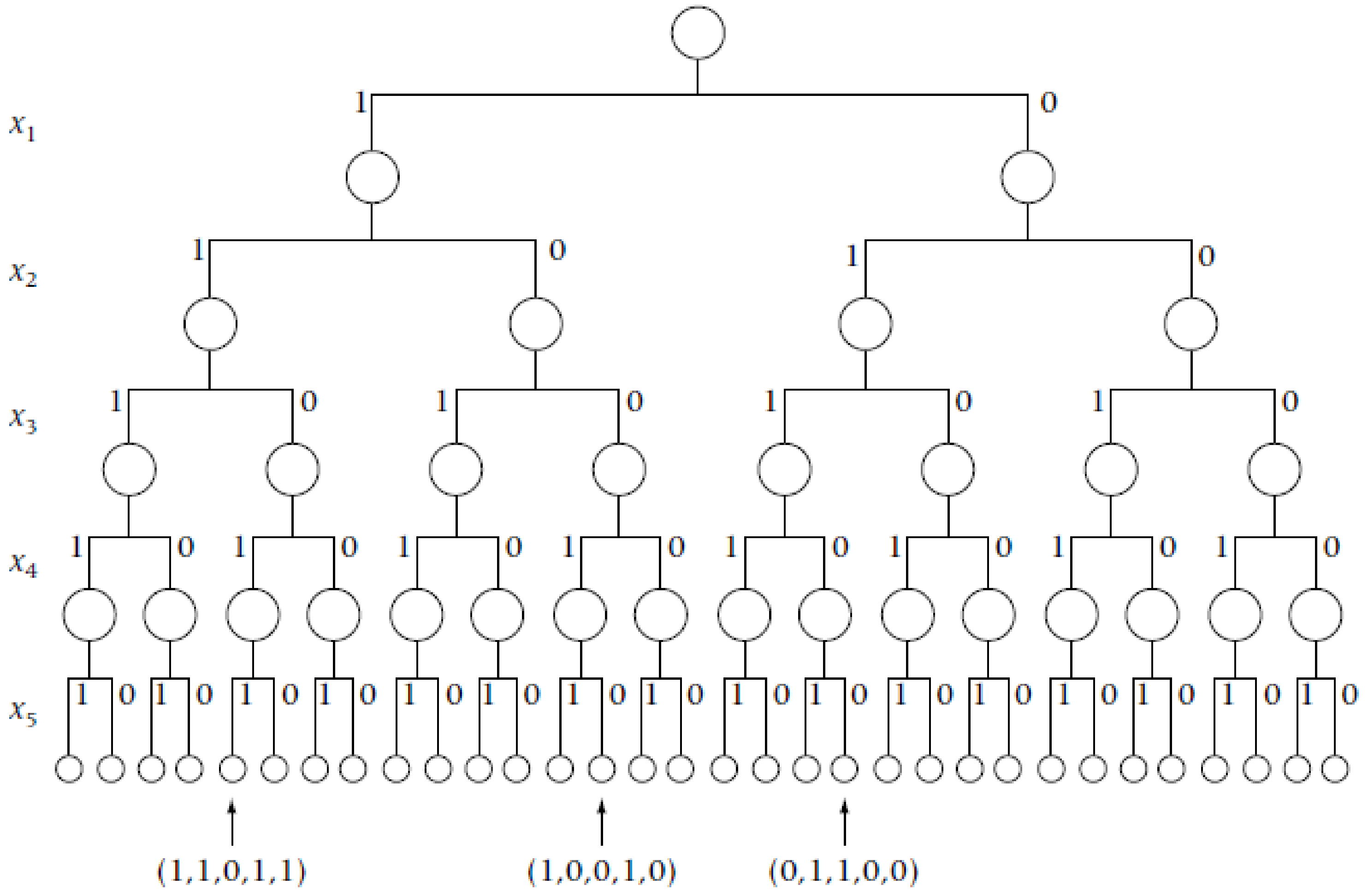
Edges labeled with index of the chosen element. Index values of problem state shown outside some sample nodes.

Fix-Tuple Model for Sum of Subsets

- In this model, the decision at stage k is whether or not to choose element a_{k-1}, $1 \le k \le n$. Thus, $D_k = \{0,1\}$, where $x_k = 1$ if element a_{k-1} is chosen, and $x_k = 0$, otherwise.
- The state space tree T associated with the decision sets $D_k(x_1,\ldots,x_{k-1})$ is the full binary tree on $2^k - 1$ nodes, with a left child of a node at level $k - 1$ corresponding to choosing a_{k-1} ($x_k = 1$) and a right child corresponding to omitting a_{k-1} ($x_k = 0$), so that

$$D_k(x_1, \ldots, x_{k-1}) = \{0,1\}, \quad 1 \le k \le n.$$

Fixed-tuple state space tree *T* for the sum of subsets problem with *n* = 5



Backtracking Strategy

- The backtracking strategy performs a depth-first search of the state space tree *T*, utilizing an appropriate bounding function.
- When a node is accessed during a backtracking search, it becomes the current node being expanded (*E-node*).
- By convention, when moving from an *E*-node to the next level of the state space tree, we select the left-most child not already visited.
- If no such child exists, or if the *E*-node is bounded, then we backtrack to the previous level.
- If only one solution to the problem is desired, then the backtracking algorithm terminates once a **goal state** is found. Otherwise, the algorithm continues until all the nodes have been exhausted, outputting each goal state when it is reached.

Backtrack Paradigm – Nonrecursive version

```
procedure Backtrack()
Input:  T (implicit state space tree associated with the given problem)
        $D_k$ (decision set, where $D_k = \emptyset$ for $k \geq n$)
        Bounded (bounding function)
Output:  all goal states
    $k \leftarrow 1$
    while $k \geq 1$ do     //E-node is (X[1],...,X[k – 1]). Initially E-node = ( )
                                               corresponding to root.
        Searching ← .true.
        while Searching do              //searching for unbounded child
            X[k] ←first of the remaining untried values from
            $D_k$(X[1],...,X[k– 1]), where this value is $\emptyset$ if all values in
            $D_k$(X[1],...,X[k – 1]) have been tried
            if X[k] ← $\emptyset$ then
                Searching ← .false.
            else
                if (X[1],..., X[k]) is a goal state then
                    Print(X[1],...,X[k])
                endif
                if .not. Bounded(X[1],...,X[k]) then
                    Searching ← .false.
                endif
            endif
        endwhile
        if X[k] = $\emptyset$ then
        Arrange for all values in $D_k$ to be considered as untried
            $k \leftarrow k - 1$              //backtrack to previous level
        else
            $k \leftarrow k + 1$              //move on to next level
        endif
    endwhile
end Backtrack
```

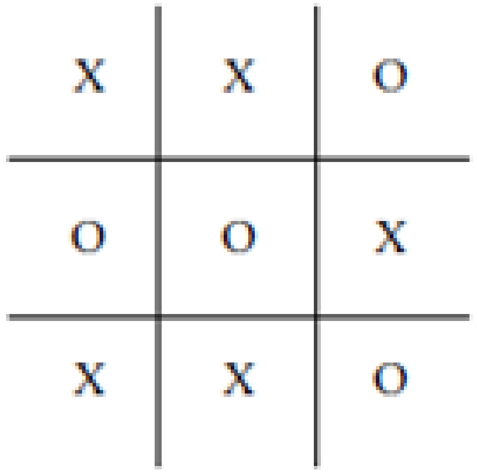
Backtrack Paradigm – Recursive version

```
procedure BacktrackRec(k) recursive
Input:  T (implicit state space tree associated with the given problem)
        k (a nonnegative integer, 0 in initial call)
        $D_k$ (decision set, where $D_k = \emptyset$ for $k \geq n$)
        $X[0{:}n]$ (global array where $X[1{:}n]$ maintains the problem states of T and
                where the problem state $(X[1],\ldots,X[k])$ has already been generated)
        Bounded (bounding function)
Output:  all goals that are descendants of $(X[1],\ldots,X[k])$
          $k \leftarrow k + 1$
          for each $x_k \in D_k(X[1],\ldots,X[k-1])$ do
              $X[k] \leftarrow x_k$
              if $(X[1],\ldots,X[k])$ is a goal state then
                      Print$(X[1],\ldots,X[k])$
              endif
              if .not. Bounded$(X[1],\ldots,X[k])$ then
                      BacktrackRec(k)
              endif
          endfor
end BacktrackRec
```

PSN

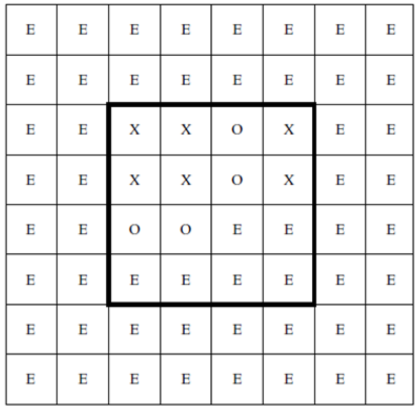
Give pseudocode for recursive version of the Sum of Subsets problem backtracking solution using the variable tuple state space tree.

3 × 3 Tic-Tac-Toe Tie Board
Cat’s Game

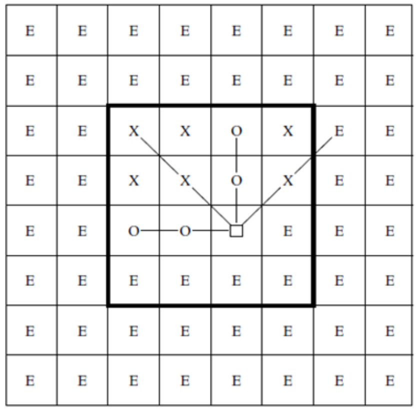


X	X	O
O	O	X
X	X	O

Convenient to extend the board



E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	X	X	O	X	E	E
E	E	X	X	O	X	E	E
E	E	O	O	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E



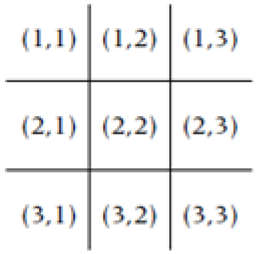
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	X	X	O	X	E	E
E	E	X	X	O	X	E	E
E	E	O	O	□	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E

Bounding Function

$$Bounded(x_1,\ldots,x_k) = \begin{cases} \textbf{.true.} & \text{if board configurat ion corresponding} \\ & \text{to } (x_1,\ldots,x_k) \text{ contains } 3 \text{ in a row.} \\ \textbf{.false.} & \text{otherwise.} \end{cases}$$

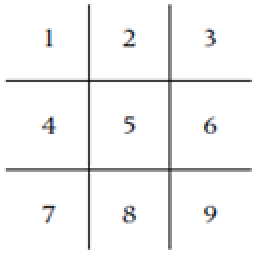
```
function BoundedBoard($i$, $j$)
Input:  $B[-1:n+2, -1:n+2]$ (global array corresponding to board configuration)
        $i, j$ (integers between 1 and $n$, inclusive)
Output: returns .true. if the board configuration involving the cells labeled
        $1,\ldots,k = n(i-1)+j$, contains three in a row in either
        Xs or Os along a line containing the cell labeled $k$.
    $LineH \leftarrow (B[i,j] = B[i,j-1])$ .and. $(B[i,j] = B[i,j]-2])$
    $LineV \leftarrow (B[i,j] = B[i-1,j])$ .and. $(B[i,j] = B[i-2,j])$
    $LineD1 \leftarrow (B[i,j] = B[i-1,j-1])$ .and. $(B[i,j] = B[i-2,j-2])$
    $LineD2 \leftarrow (B[i,j] = B[i-1,j+1])$ .and. $(B[i,j] = B[i-2,j+2])$
    return($LineH$ .or. $LineV$ .or. $LineD1$ .or. $LineD2$)
end BoundedBoard
```

Representation of board positions Generalizes to $n \times n$ board



(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

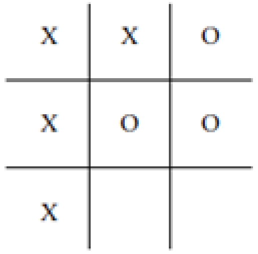
(a)



1	2	3
4	5	6
7	8	9

(b)

row-major order



X	X	O
X	O	O
X		

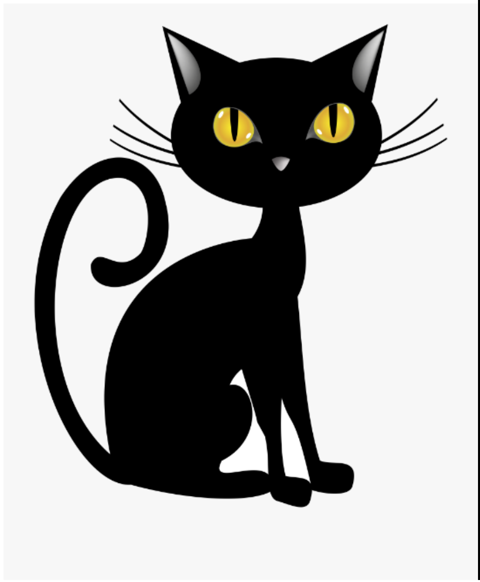
(c)

Next Position of $n \times n$ Tic-Tic-Toe Board

Next(i,j) is the next in **row-major order** implemented as follows:

```
if $j < n$ then $j \leftarrow j + 1$  //next column in same row
         else   //first column in next row
        $i \leftarrow i + 1$
        $j \leftarrow 1$
endif
```

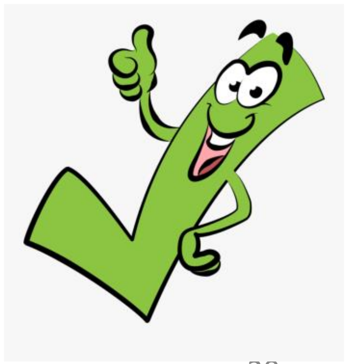
Pseudocode for backtracking solution generating all cat's games for $n \times n$ Tic-Tic-Toe



```
procedure TicTacToeRec(i,j) recursive
Input:  i, j (integers between 1 and n, inclusive, called initially with i = 0
                                                        and j = n)
        B[– 1:n + 2, – 1:n + 2]   (global array corresponding to board
                      configuration, initialized to 'E', and B[1,1],...,B[i,j] filled
                      with Xs and Os with no three in a row)
Output:  all extensions of B[1,1],...,B[i,j] to goal states; that is, board
                    configurations not containing three in a row in either Xs or Os
          Next(i,j)              //k = k + 1
          for Child ← 1 to 2 do
              if Child = 1 then
                      B[i,j] ← 'X'
              else
                      B[i,j] ← 'O'
              endif
              if .not. BoundedBoard(i,j) then
                      if (i = n) .and. (j = n) then
                              PrintBoard(Board[1:n,1:n])   //print goal state
                      else
                              TicTacToeRec(i,j)
                      endif
              endif
          endfor
end TicTacToeRec
```

Correctness

- In a cat’s game, i.e., tie board, the number of Xs and Os differ by at most one.
- It is interesting that all tie boards in the n n board where there are no “three in a row” have this property, so we don’t need to check for it in our backtracking algorithm.
- Therefore, our algorithm is correct!



Branch-and-Bound

- As with backtracking algorithms, branch-and-bound algorithms are based on searches of an associated state space tree for goal states.
- However, in a branch-and-bound algorithm, **all** the children of the *E*-node (the node currently being expanded) are generated before the next *E*-node is chosen.
- When the children are generated, they become **live** nodes and are stored in a suitable data structure *LiveNodes*.
- *LiveNodes* is typically a queue, a stack, or a priority queue corresponding to *FIFO* (*First-In, First-Out*) *branch-and-bound*, *LIFO* (*Last-In*, *First-Out*) *branch-and-bound*, and *least cost branch-and-bound*, respectively.

Branch-and-Bound

- Immediately upon expanding the current *E*-node, this *E*-node becomes a *dead* node and a new *E*-node is selected from *LiveNodes*.
- Branch-and-bound is quite different from backtracking, where we might backtrack to a given node many times, making it the *E*-node each time until all its children have finally been generated or the algorithm terminates.
- The nodes of the state space tree at any given point in a branch-and-bound algorithm are in one of the following four states: *E-node, live node, dead node,* or *not yet generated*.

Bounding Function

- As with backtracking, the efficiency of branch-and-bound depends on the utilization of good bounding functions.
- Such functions are used in the attempt to determine solutions by restricting attention to small portions of the entire state space tree.
- When expanding a given *E*-node, a child can be bounded if it can be shown that it cannot lead to a goal node.

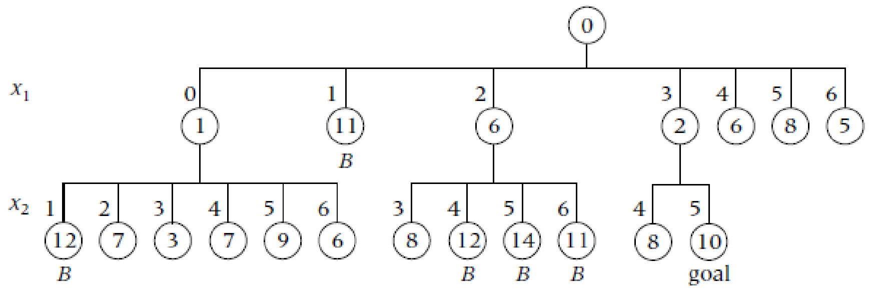
FIFO Branch-and-Bound

- A **FIFO branch-and-bound** involves performing a breadth-first search of the state space tree, i.e., data structure *LiveNodes* is a **queue**.
- Initially the queue of live nodes is empty.
- The algorithm begins by generating the root node of the state space tree and enqueuing it in the queue *LiveNodes*.
- At each stage of the algorithm a node is dequeued from *LiveNodes* to become the new *E*-node.
- All the children of the *E*-node are then generated.
- The children that are not bounded are enqueued.
- If only one goal state is desired, then the algorithm terminates after the first goal state is found. Otherwise, the algorithm terminates when *LiveNodes* is empty.

Action of FIFO Branch-and-Bound for an example Sum of Subsets instance

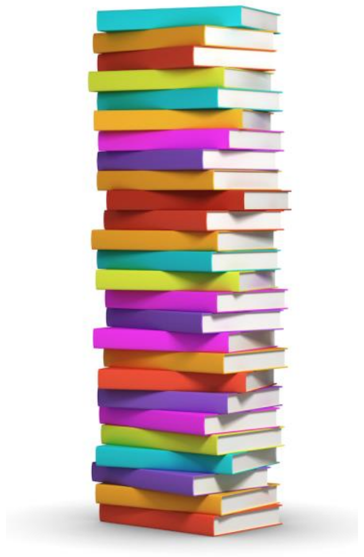
Action of queue *LiveNodes* and a portion of the variable-tuple state space tree generated by FIFO branch-and-bound for the sum of subsets problem with $A = (1,11,6,2,6,8,5)$ and $Sum = 10$. The sum of the elements chosen is shown inside each node.

	queue *LiveNodes*
generate ()	enqueue ()
dequeue	*E*-node = ()
generate (0)	enqueue (0)
generate (1)	bounded
generate (2)	enqueue (2)
generate (3)	enqueue (3)
generate (4)	enqueue (4)
generate (5)	enqueue (5)
generate (6)	enqueue (6)
dequeue	*E*-node = (0)
generate (0,1)	bounded
generate (0,2)	enqueue (0,2)
generate (0,3)	enqueue (0,3)
generate (0,4)	enqueue (0,4)
generate (0,5)	enqueue (0,5)
generate (0,6)	enqueue (0,6)
dequeue	*E*-node = (2)
generate (2,3)	enqueue (2,3)
generate (2,4)	bounded
generate (2,5)	bounded
generate (2,6)	bounded
dequeue	*E*-node = (3)
generate (3,4)	enqueue (3,4)
generate (3,5)	goal node



LIFO Branch-and-Bound

LIFO branch-and-bound is similar to FIFO branch-and-bound except *LiveNodes* is a **stack** instead of a queue.

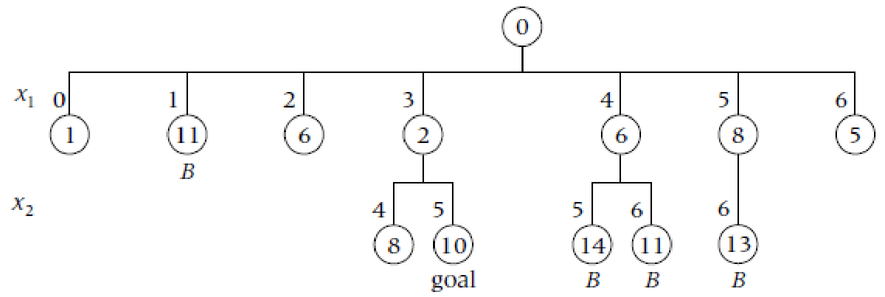


Action of LIFO Branch-and-Bound for an example Sum of Subsets instance

Action of queue *LiveNodes* and a portion of the variable-tuple state space tree generated by LIFO branch-and-bound for the sum of subsets problem with *A* = (1,11,6,2,6,8,5) and *Sum* = 10. The sum of the elements chosen is shown inside each node.

stack *LiveNodes*

generate ()	push ()
pop	*E*-node := ()
generate (0)	push (0)
generate (1)	bounded
generate (2)	push (2)
generate (3)	push (3)
generate (4)	push (4)
generate (5)	push (5)
generate (6)	push (6)
pop	*E*-node := (6)
pop	*E*-node := (5)
generate (5,6)	bounded
pop	*E*-node := 4
generate (4,5)	bounded
generate (4,6)	bounded
pop	*E*-node := (3)
generate (3,4)	push (3,4)
generate (3,5)	goal node



General Branch-and-Bound Paridigm

```
procedure BranchAndBound
Input:  function $D_k(x_1,...,x_{k-1})$ determining state space tree $T$ associated with the
                                                                given problem)
          Bounding function Bounded
Output:  All goal states to the given problem
        LiveNodes is initialized to be empty
        AllocateTreeNode(Root)
        Root $\rightarrow$ Parent $\leftarrow$ null
        Add(LiveNodes,Root)                  //add root to list of live nodes
   while LiveNodes is not empty do
            Select(LiveNodes,E-node,k)      //select next E-node from live nodes
        for each $X[k] \in D_k$(E-node) do          //for each child of the E-node do
            AllocateTreeNode(Child)
            Child $\rightarrow$ Info $\leftarrow$ $X[k]$
            Child $\rightarrow$ Parent $\leftarrow$ E-node
            if Answer(Child) then              //if child is a goal node then
                Path(Child)                    //output path from child to root
            endif
            if .not. Bounded(Child) then
                Add(LiveNodes,Child)      //add child to list of live nodes
            endif
        endfor
   endwhile
end BranchAndBound
```

Utilizing Heuristics

- Both LIFO and FIFO branch-and-bound are blind searches of the state space tree *T* in the sense that they search the nodes of *T* in the same order regardless of the input to the algorithm. Thus, they tend to be inefficient for searching the large state space trees that often arise in practice.
- Utilizing heuristics can help narrow the scope of otherwise blind searches.
- For example, the **least cost branch-and-bound** strategy utilizes a heuristic cost function associated with the nodes of the state space tree *T*, where the set of live nodes is maintained as a **priority queue** with respect to this cost function.
- In this way, the next node to become the *E*-node is the one that is the most promising to lead quickly to a goal.

The left-hand backtracking algorithm versus heuristic search
JIM BORGMAN, *The Cincinnati Enquirer*.