Back Tracking

Back tracking can be described as an organized exhaustive search which often avoids searching all possibilities.

This technique is generally suitable for solving problems where a potentially large, but finite number of solutions have to be inspected.

Here the desired solution is to express as an n- tuple $(x_1,x_2,x_3 \ldots x_n)$ where the x_i are chosen from some finite S_i.

If m_i is the size of set S_i. Then there are $m = m_1 m_2 \ldots m_n$ n tuples that are possible candidates.

The back tracking algorithm has its virtue the ability to yield the same answer with far fewer than m trials.

Its basic idea is to build up the solution vector one component at a time and

to use some criterion function $P_i(x_1, x_2, \ldots x_i)$ to test whether the vector being formed has any chance of success.

If it is realized that partial vector ($x_1, x_2 \ldots x_i$) can in no way lead to an optimal solution, then $m_{i+1} m_{i+2} \ldots m_n$ possible test vectors can be ignored entirely.

Some terms used in back tracking approach

Explicit Constraints: these are rules that restrict each x_i to take values only from a given set.

e.g. $X_i >= 0$

x_i=0 or 1

Implicit constraints: These are rules which describes the way in which the x_i must relate to each other.

Solution space: it is the set of all tuples that satisfy the explicit constraints.

State space tree: The solution space is organized as a tree called state space tree.

Live node: A node which has been generated and all of whose children have not been generated is called a live node.

Dead node: it is a generated node which is not to be expanded further or all of whose children have been generated

E- node: the live node whose children are currently being generated is called E-node.

Bounding function: is a function created which is used to kill live nodes without generating all its children.

The Bicycle problem

Consider a combination lock for a bicycle that consists of a set of N switches each of which can be ‘on’ or ‘off’.

Exactly one setting of all the switches with [N/2] or more in the ‘on’ position, will open the lock.

Suppose we have forgotten this combination and must get the lock open.

Suppose also that you are willing to try (if necessary) all combinations.

We need an algorithm for systematically generating these combinations.

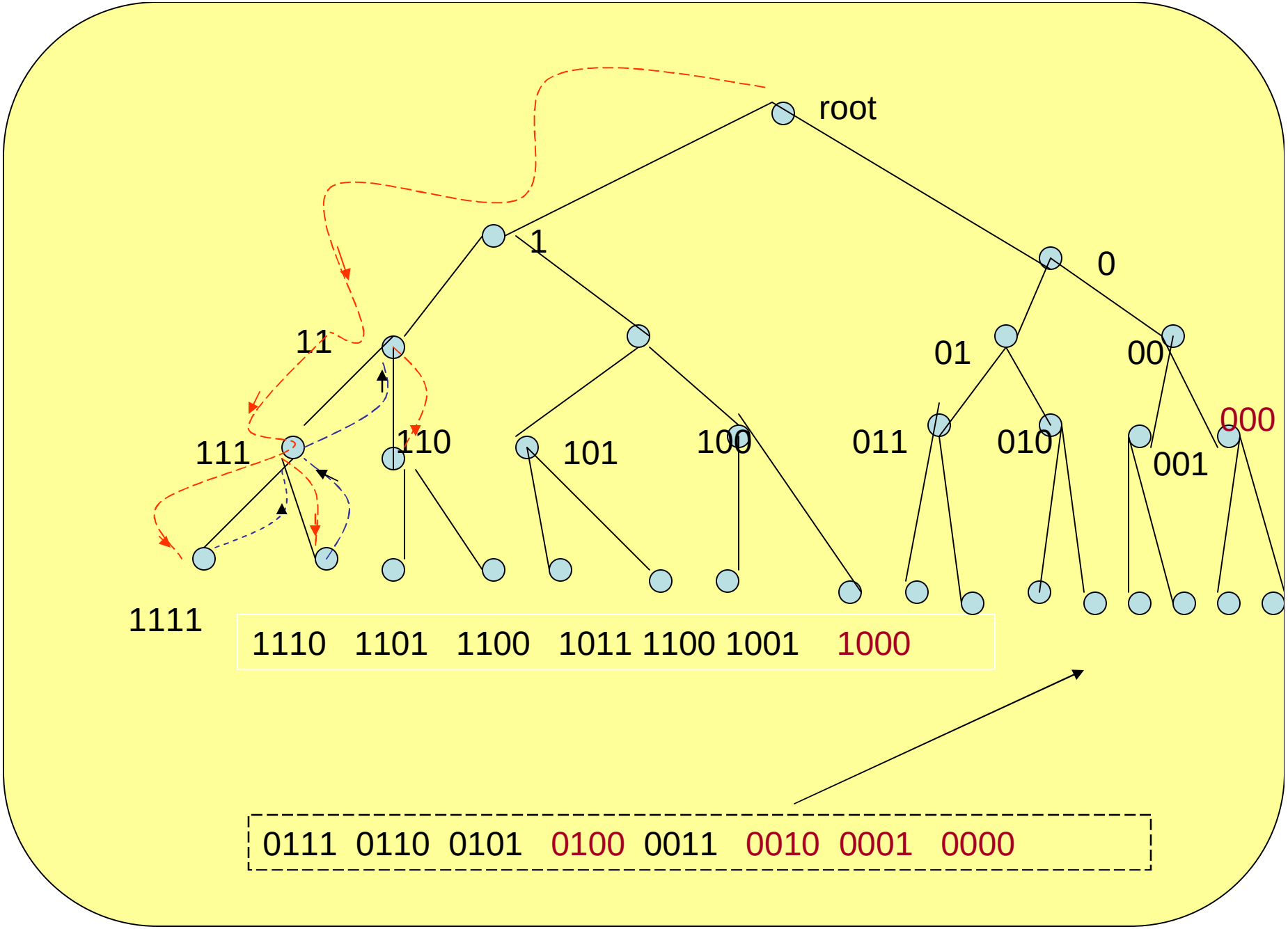
If we ignore the n/2 condition(given clue), there are 2^n possible combinations for the lock.

We model each possible combination with an n- tuple of 0's and 1's.

1 for the switch on

0 for switch off.

We can represent all combinations as binary tree.


root
1
0
11
01
00
111
110
101
100
011
010
001
000
1111
1110 1101 1100 1011 1100 1001 1000
0111 0110 0101 0100 0011 0010 0001 0000

1 →11 →111 →1111(check) →111(backtr)
→1110(check) →111(backtr) →11(backtr)
→110→1101(check) →110(backtr)
→1100 →110(backtr) →11(backtr)
→1(backtr.)→10 →101 →1011(check)
→101(back tr.) →1010(check) →101
(back tr.) →10(back tr.) →100
→1001(check) →100 (back tr.) →10(back
tr.) →1 (back tr.) →root →0….

Using this binary tree we can now state a back track procedure for generating only those combinations with atleast [n/2] switches on.

This algorithm amounts to traversal of the tree.

Move down the tree as far as possible to the left until we can move no further.

Upon reaching an end vertex, try the corresponding combination.

If it fails, back track up one level and see if we can move down again along an unsearched branch.

If it is possible take the leftmost unused branch.

If not back track up one more level and try to move down from this vertex.

Before moving down check if it is possible to satisfy the [n/2] restriction at any successor vertex

Algorithm terminates when we return to the root and there are no unused branches remaining.

8 – Queen Problem

To place eight queens on an 8 by 8 chess board so that no two " attack" i.e. no two queens of them are on the same row, column ,or diagonal.

To solve it by back tracking approach we must see the solution as an n- tuple $(x_1,\ldots.x_n)$ where each x_i is coming from a finite set.

Let us number the queens 1 to 8.

Also number rows and columns from 1 to 8.

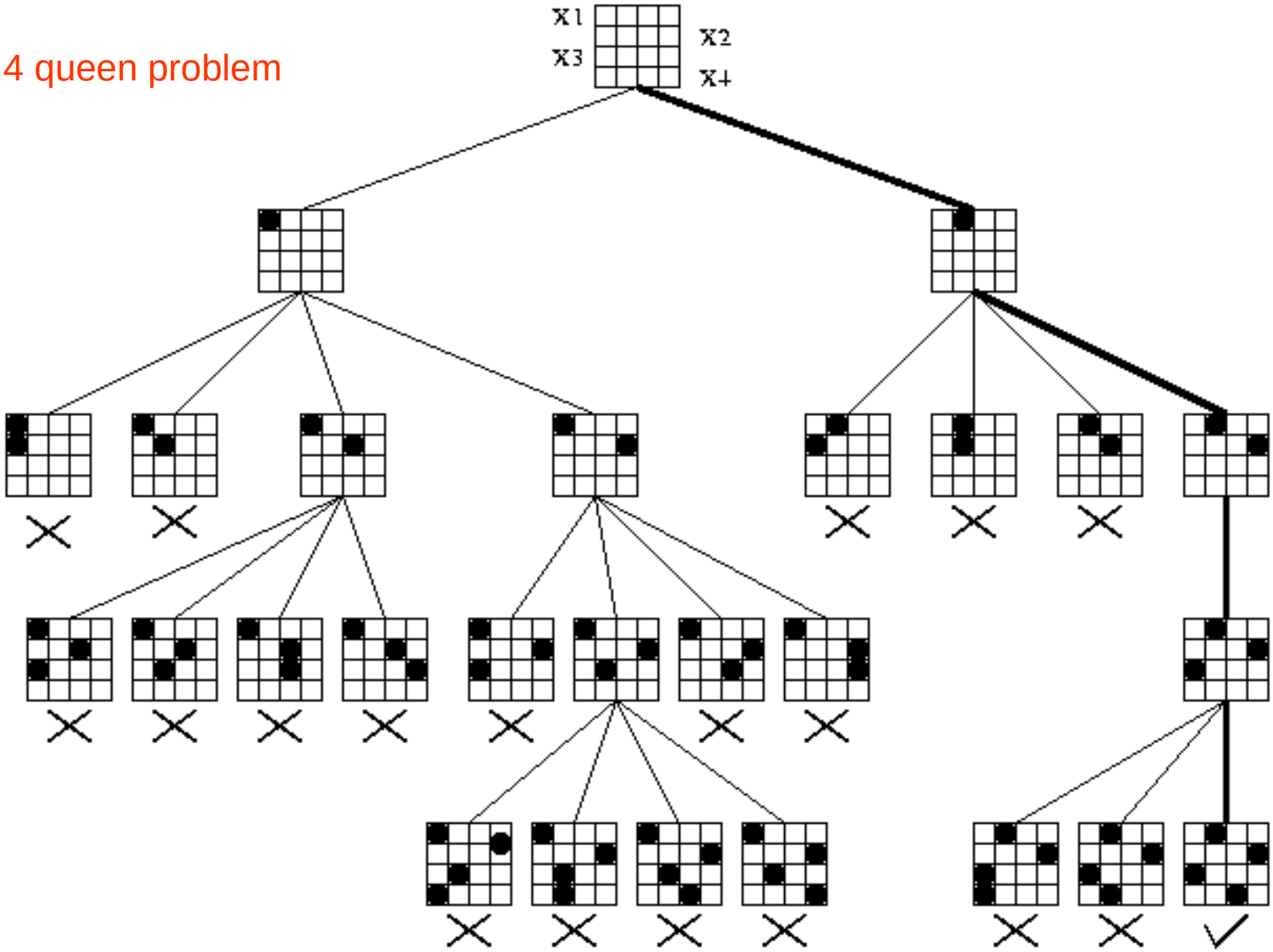
Since each queen has to be in different row we can assume that queen 1 moves in row 1 , queen 2 moves in row 2 and so on.

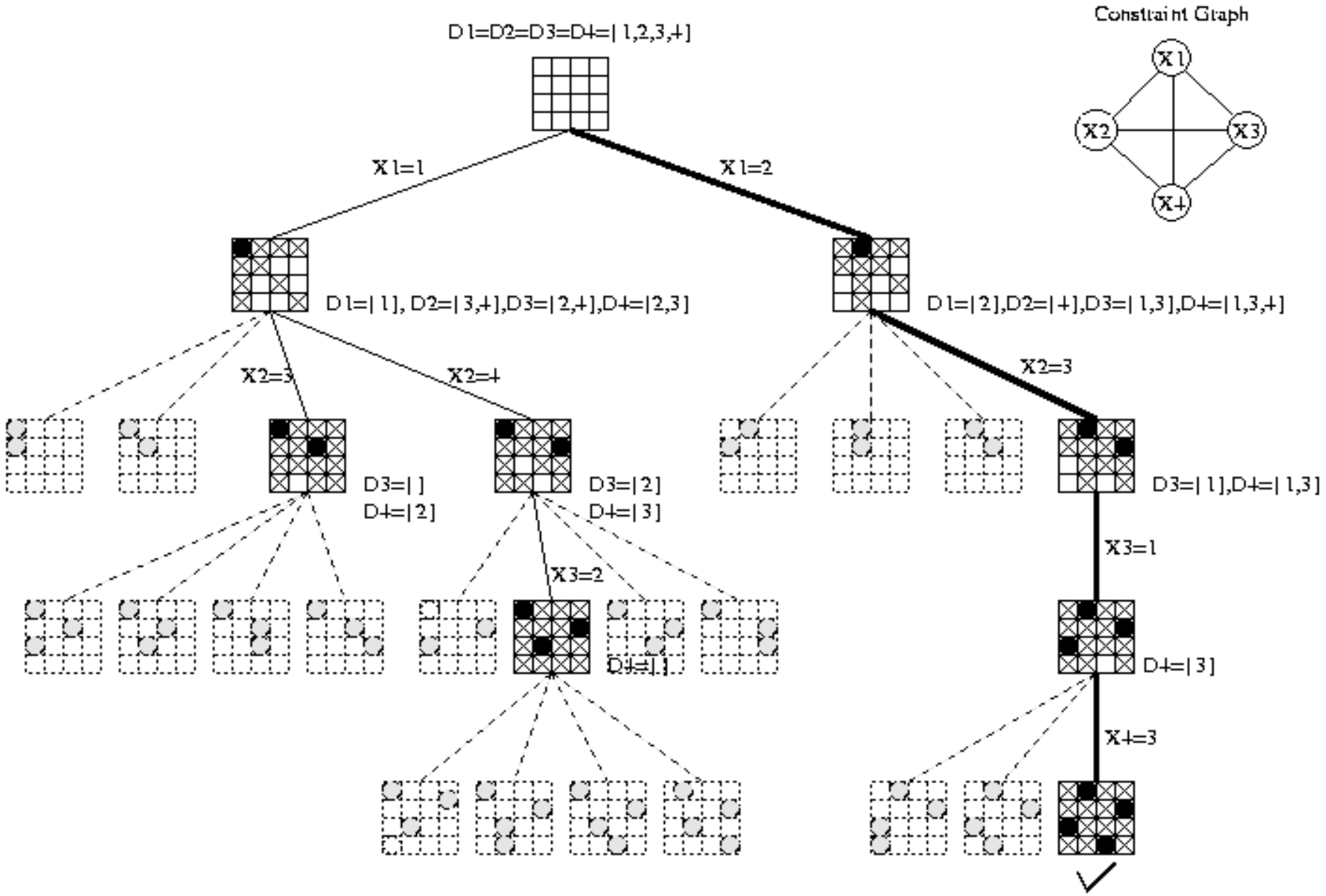
Using this convention,

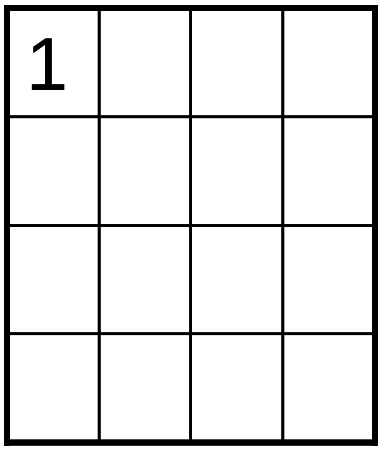
Any non attacking placement of queens can be represented as (c_1 , c_2, c_3……c_8)

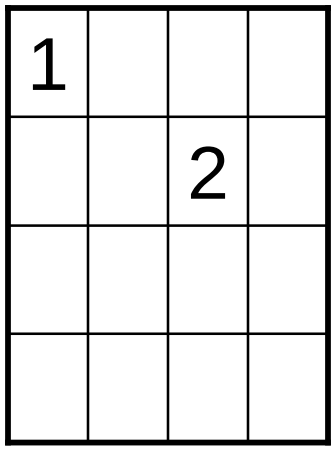
C_i represents the column number in which queen i is placed.

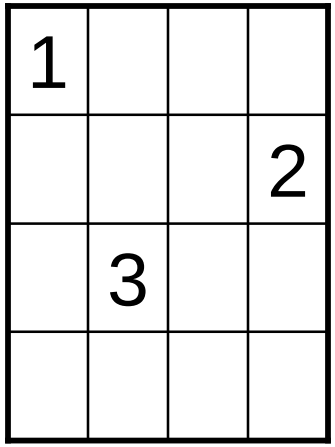
4 queen problem

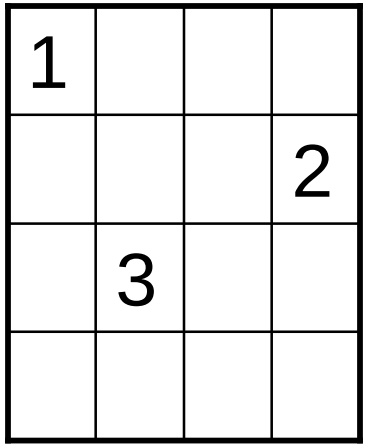


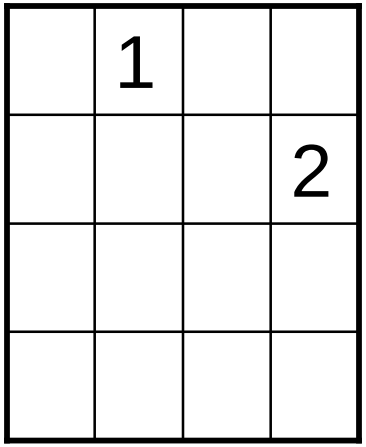

Constraint Graph
X1
X2
X3
X4
D1=D2=D3=D4=|1,2,3,4|
X1=1
X1=2
D1=|1|, D2=|3,4|,D3=|2,4|,D4=|2,3|
D1=|2|,D2=|4|,D3=|1,3|,D4=|1,3,4|
X2=3
X2=4
X2=3
D3=|1|
D4=|2|
D3=|2|
D4=|3|
D3=|1|,D4=|1,3|
X3=1
X3=2
D4=|1|
D4=|3|
X4=3

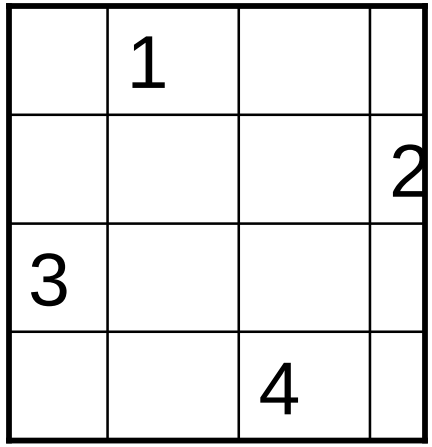

1

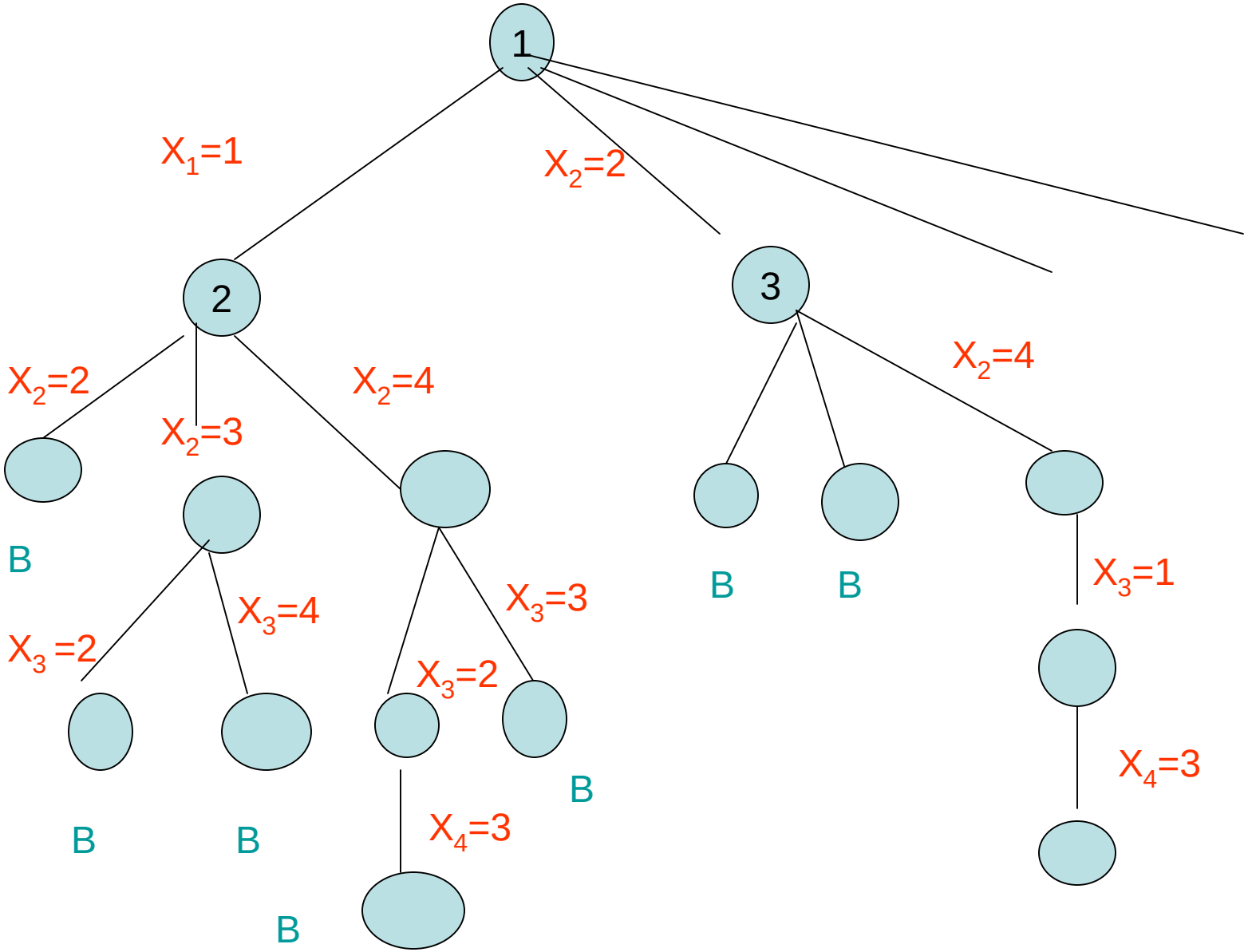

1
2


1
2
3


1
2
3


1
2


1
2
3
4


1
X_1=1
X_2=2
2
3
X_2=2
X_2=3
X_2=4
X_2=4
B
X_3=4
X_3=3
X_3 =2
X_3=2
X_3=1
B
B
B
B
B
X_4=3
X_4=3
B

Sum of subsets problem

Suppose we are given n distinct positive numbers and we desire to find all combinations of these numbers whose sums are m.

.

- Case Study 2: Sum of Subsets (cont.)
- The following incarnation *SuM* of the general backtracking algorithm
- has two additional formal parameters:
- • *s* for the sum of elements that has been selected,
- • *r* for the overall sum of the remaining stock.
- **Algorithm** *Backtracking(k)*
- { for each child *x[k]* of *x[k-1]* in T do
- if *x[1..k]* does not meet the boundary B
- then {if *x[1..k]* is a path to a leaf of T
- then { if the leaf meets the implicit constraint C
- then output the leaf and the path *x[1..k]*
- }
- else Backtracking(*k*+1)
- }
- }
- **Algorithm** *SuM*(*k, s, r*)
- { *x*[*k*]:=1 **;**
- if (*s*<=M) & (*r*>=M-*s*) & (*w*[*k*]<=M-*s*)
- then { if M = *s* +*w*[*k*]
- then output all *i*Î[1..*k*] that *x*[*i*]=1
- else *SuM*(*k*+1, *s*+*w*[*k*], *r* -*w*[*k*])
- } **;**
- *x*[*k*]:=0 **;**
- if (*s*<=M) & (*r*>=M-*s*) & (*w*[*k*]<=M-*s*)
- then { if M = *s*
- then output all *i*Î[1..*k*] that *x*[*i*]=1
- else *SuM*(*k*+1, *s*, *r* -*w*[*k*])
- }
- }

Case Study 2: Sum of Subsets

Sum of Subset Problem:

For given N-element vector of non-decreasing positive weights w[1..N] and integer number M generate all vectors $i_1 <... < i_n$ of indexes that

$$M = \sum_{j=1}^{j=n} w[i_j] .$$

We are going to specialize the recursive algorithm *Backtracking(k)*. For it we must define the state space tree T and the boundary B.

```
Algorithm Backtracking(k)
{ for each child x[k] of x[k-1] in T do
      if x[1..k] does not meet the boundary B
      then {if x[1..k] is a path to a leaf of T
            then { if the leaf meets the implicit constraint C
                   then output the leaf and the path x[1..k]
                   }
            else Backtracking(k+1)
            }
}
```

Case Study 2: Sum of Subsets (cont.)

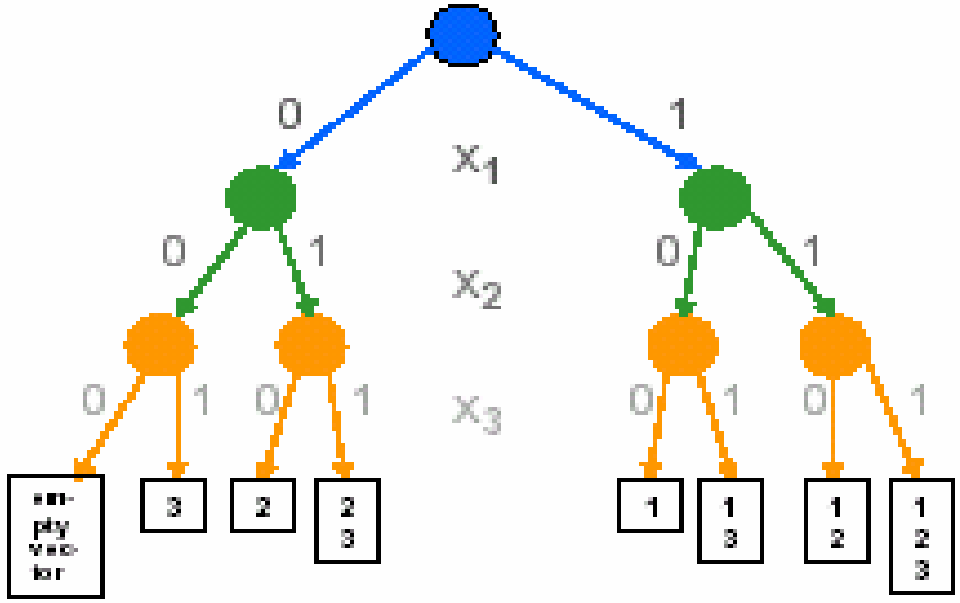
Every vector of indexes $i_1 < \ldots < i_n$ in [1..N] can be presented by its characteristic vector

indexes = 1.. (i_1-1) i_1 (i_1+1) ... (i_n-1) i_n (i_n+1) .. N

$x = (0 \ldots 0 \ 1 \ 0 \ \ldots . \ 0 \ 1 \ 0 \ \ldots 0)$,

where 1's are in positions indexed by $i_1 < \ldots < i_n$.

In this settings the state space tree T becomes the complete binary tree of height N:



Case Study 2: Sum of Subsets (cont.)

Boundary

1. Sum of the selected elements in $w[1],...w[n]$ ($n<=N$) exceeds M:
   $s = \sum_{x_i=1 \wedge i \in [1..n]} w[i] = \sum_{i=1}^{i=n} x_i \times w[i] > M.$

2. $s<M$ but remaining stock is $w[n+1], \ldots w[N]$ can not complete M:
   $r = \sum_{i=n+1}^{i=N} w[i] < (M - s).$

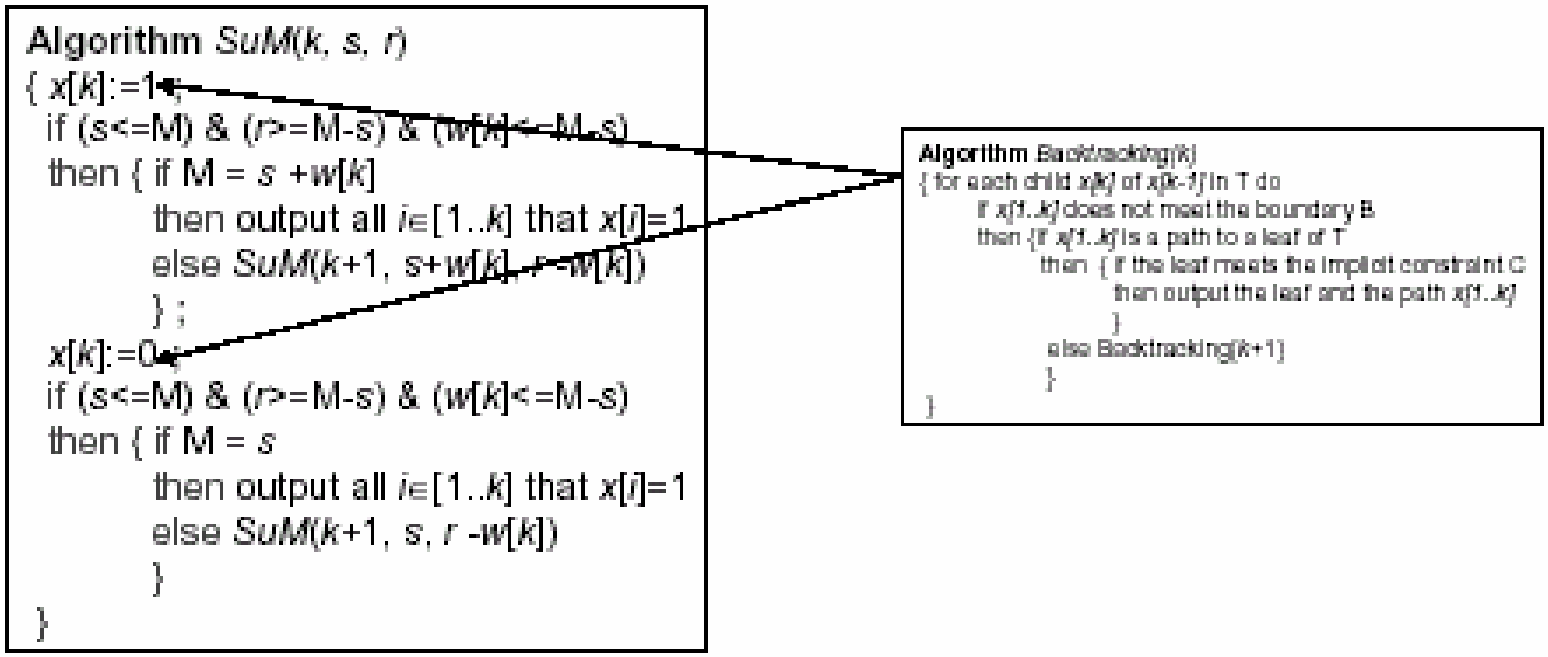
3. $s<M$ but the minimal value in the remaining stock $w[k+1], \ldots w[N]$ overflows M: $w[n+1] > (M - s)$.

The boundary B is disjunction of this three conditions B_1, B_2, and B_3.

Case Study 2: Sum of Subsets (cont.)

The following incarnation *SuM* of the general backtracking algorithm has two additional formal parameters:

- *s* for the sum of elements that has been selected,
- *r* for the overall sum of the remaining stock.



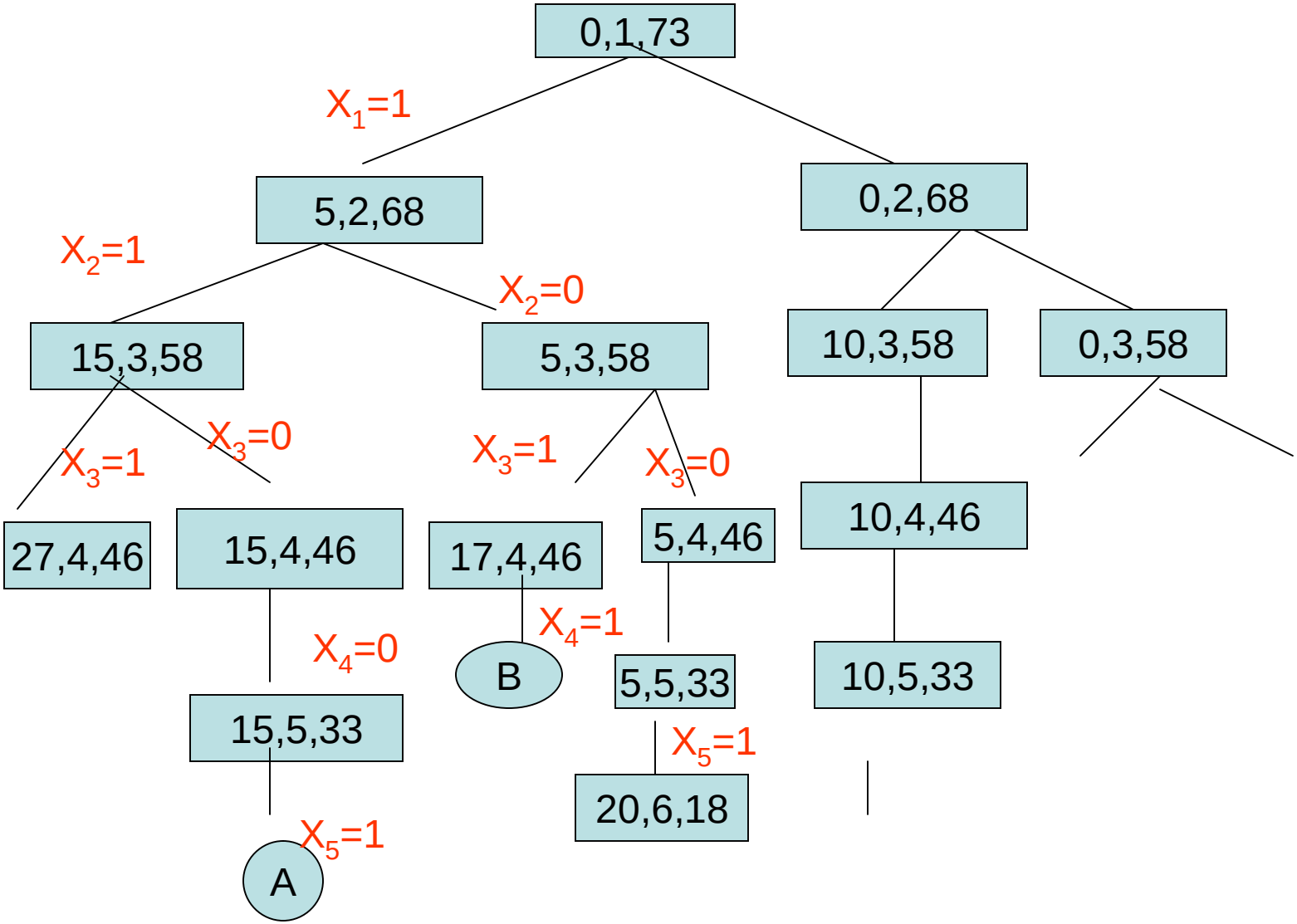
Case Study 2: Sum of Subsets (cont.)

The following algorithm *ReSuM*(*k*, *s*, *r*) optimizes *SuM*(*k*, *s*, *r*).

```
Algorithm ReSuM(k, s, r)
{ x[k]:=1 ;
  if (r>=M-s) & (w[k]<=M-s)
  then { if M = s +w[k]
         then output all i∈[1..k] that x[i]=1
         else ReSuM(k+1, s+w[k], r -w[k])
         } ;
  x[k]:=0 ;
  if (r>=M-s) & (w[k]<=M-s)
  then ReSuM(k+1, s, r -w[k])
 }
```

Example

- W[1..6] ={5, 10, 12, 13, 15, 18}
- M=30


0,1,73
X1=1
5,2,68
0,2,68
X2=1
X2=0
15,3,58
5,3,58
10,3,58
0,3,58
X3=1
X3=0
X3=1
X3=0
27,4,46
15,4,46
17,4,46
5,4,46
10,4,46
X4=0
X4=1
B
5,5,33
10,5,33
15,5,33
X5=1
20,6,18
X5=1
A

0-1 Knapack Problem

Given n positive weights w_i, n positive profits p_i and a positive number M which is the knapsack capacity

The problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le M \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i$$

is maximized

Two possible tree organization is possible.

One corresponds to the fixed tuple size formulation and the other to the variable tuple size formulation.

A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtained by expanding the given live node and any of its descendants.

If this upper bound is not higher than the value of the best solution determined so far then this live node may be killed.

Using fixed tuple size formulation:

If at a node Z the values of x_i , $1 \le i \le k$ have already been determined, then an upper bound for Z can be obtained by relaxing the requirement

x_i=0 or 1 to $0 \le x_i \le 1$ for $k+1 \le i \le n$

And it can be solved by simple greedy method

Example: P=(11,21,31,33,43,53,55,65)

W= (1,11,21,23,33,43,45,55)

M = 110

N= 8

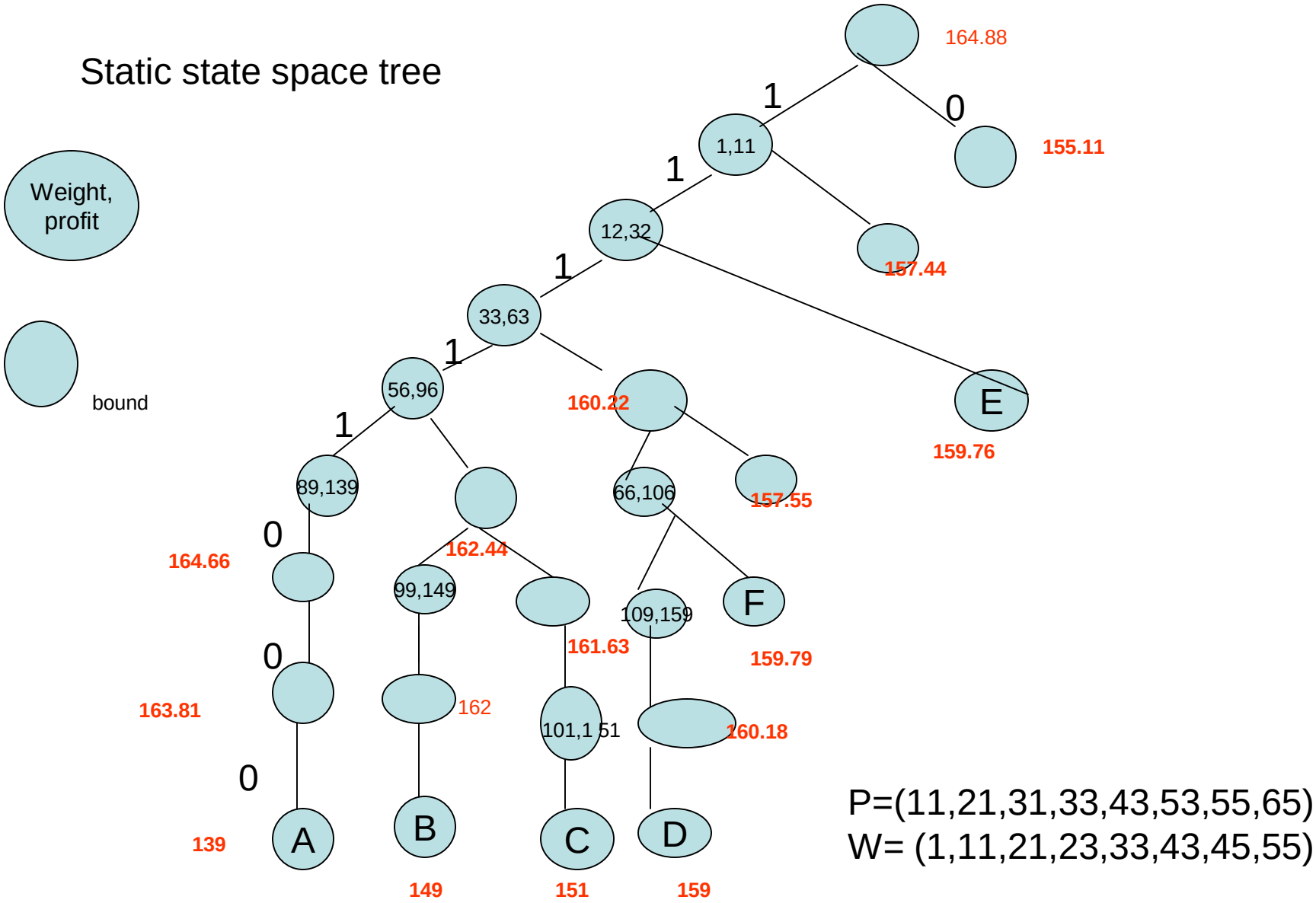
Relaxing 0,1 condition and using greedy approach

solution is (1,1,1,1,1,21/43, 0, 0)

Having 164.88 profit.

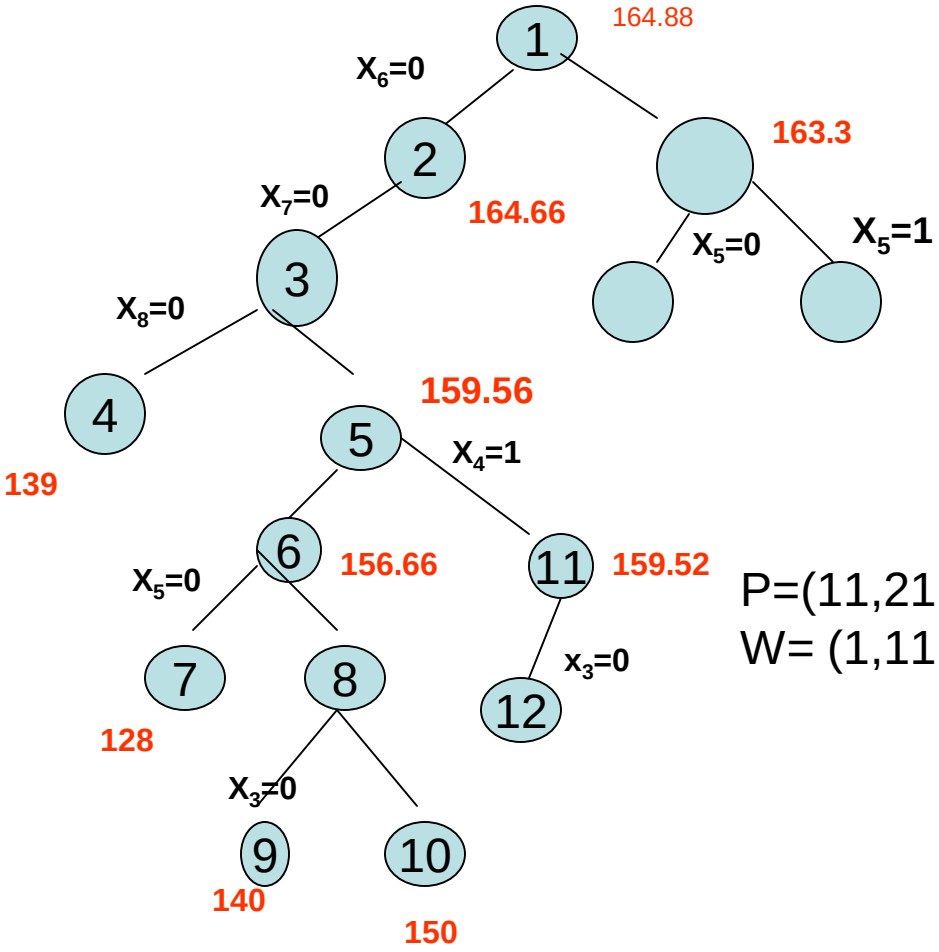
This value will work as an upper bound for the required solution.

Static state space tree



P=(11,21,31,33,43,53,55,65)
W= (1,11,21,23,33,43,45,55)

Dynamic state space tree for knapsack problem



P=(11,21,31,33,43,53,55,65)
W= (1,11,21,23,33,43,45,55)