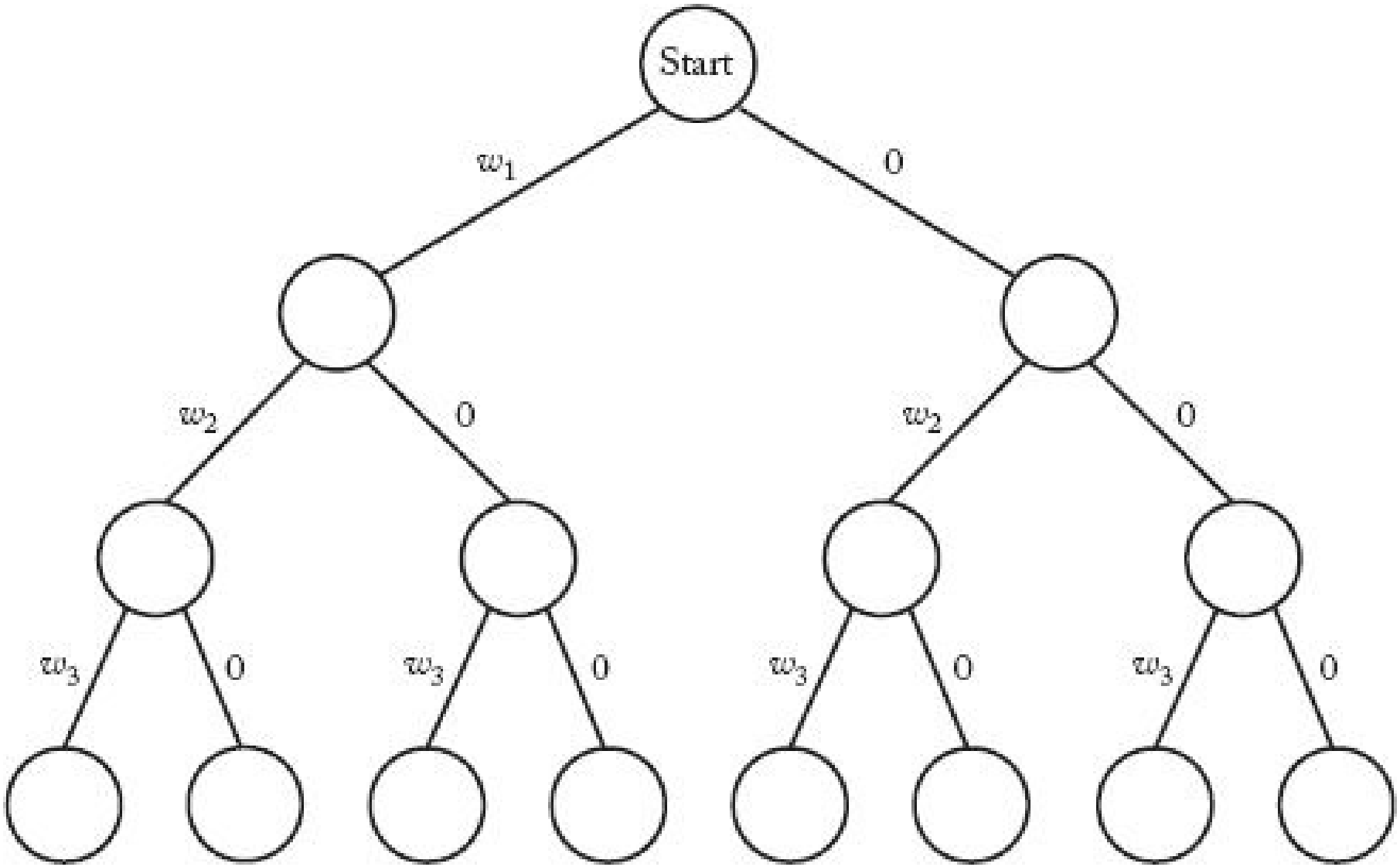
بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

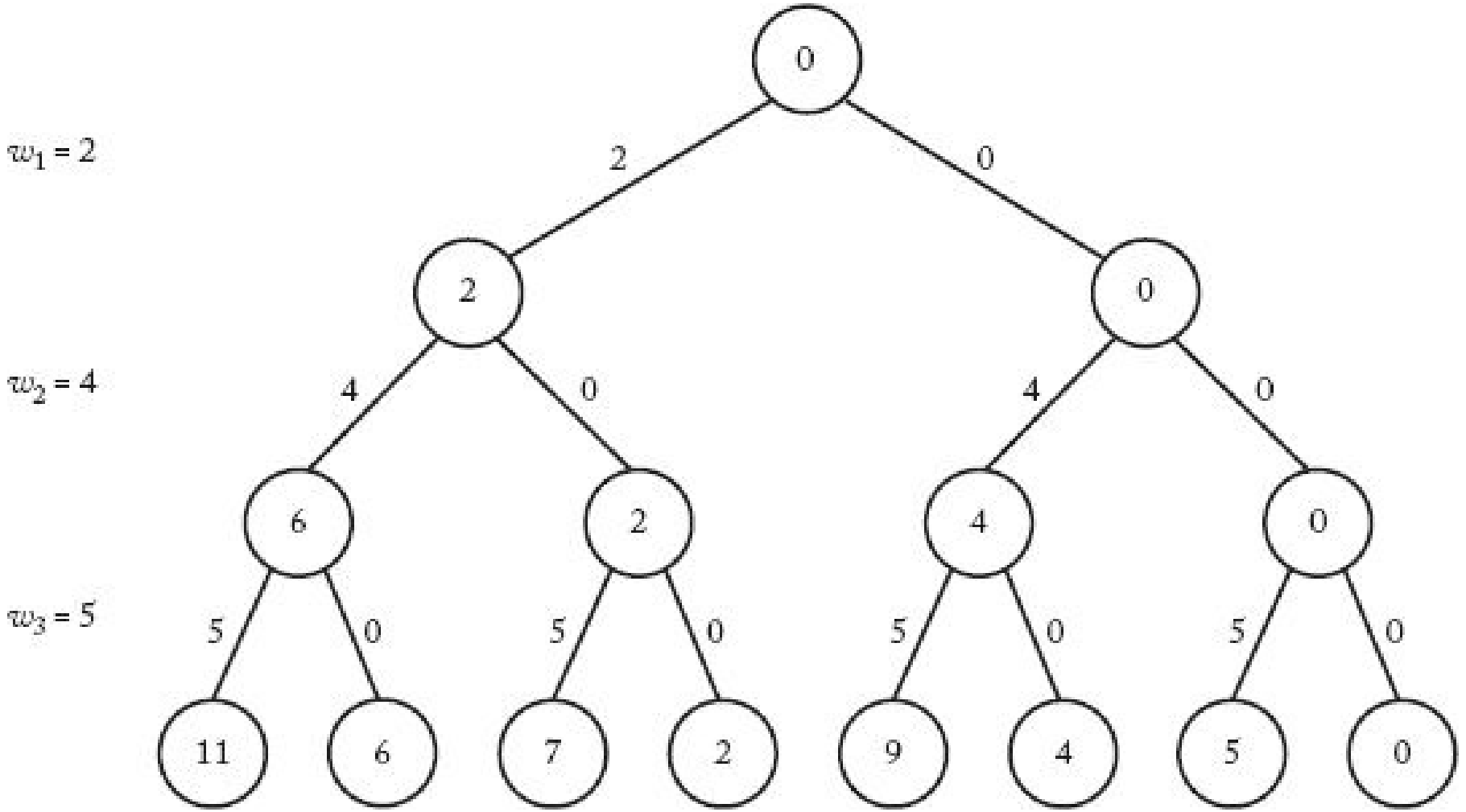
طراحی الگوریتم‌ها

حسین فلسفین

An small instance can be solved by inspection. For larger values of n, a systematic approach is necessary. One approach is to create a state space tree. A possible way to structure the tree appears in the following figure. For the sake of simplicity, the tree in this figure is for only three weights. We go to the left from the root to include w_1, and we go to the right to exclude w_1. Similarly, we go to the left from a node at level 1 to include w_2, and we go to the right to exclude w_2, etc. Each subset is represented by a path from the root to a leaf. When we include w_i, we write w_i on the edge where we include it. When we do not include w_i, we write 0.


Start
w_1
0
w_2
0
w_2
0
w_3
0
w_3
0
w_3
0
w_3
0

The state space tree for $n = 3, W = 6,$ *and* $\{w_1 = 2, w_2 = 4, w_3 = 5\}$



نشانهٔ اول برای غیرامیدبخش بودن

If we sort the weights in **nondecreasing order** *before doing the search, there is an obvious* **sign** *telling us that a node is nonpromising. If the weights are sorted in this manner, then* w_{i+1} *is the* **lightest weight remaining** *when we are at the* i*th level. Let* $weight$ *be the sum of the weights that have been included up to a node at level* i*. If* w_{i+1} *would bring the value of weight above* W*, then so would any other weight following it. Therefore, unless weight equals* W *(which means that there is a solution at the node), a node at the* i*th level is nonpromising if*

$$weight + w_{i+1} > W.$$

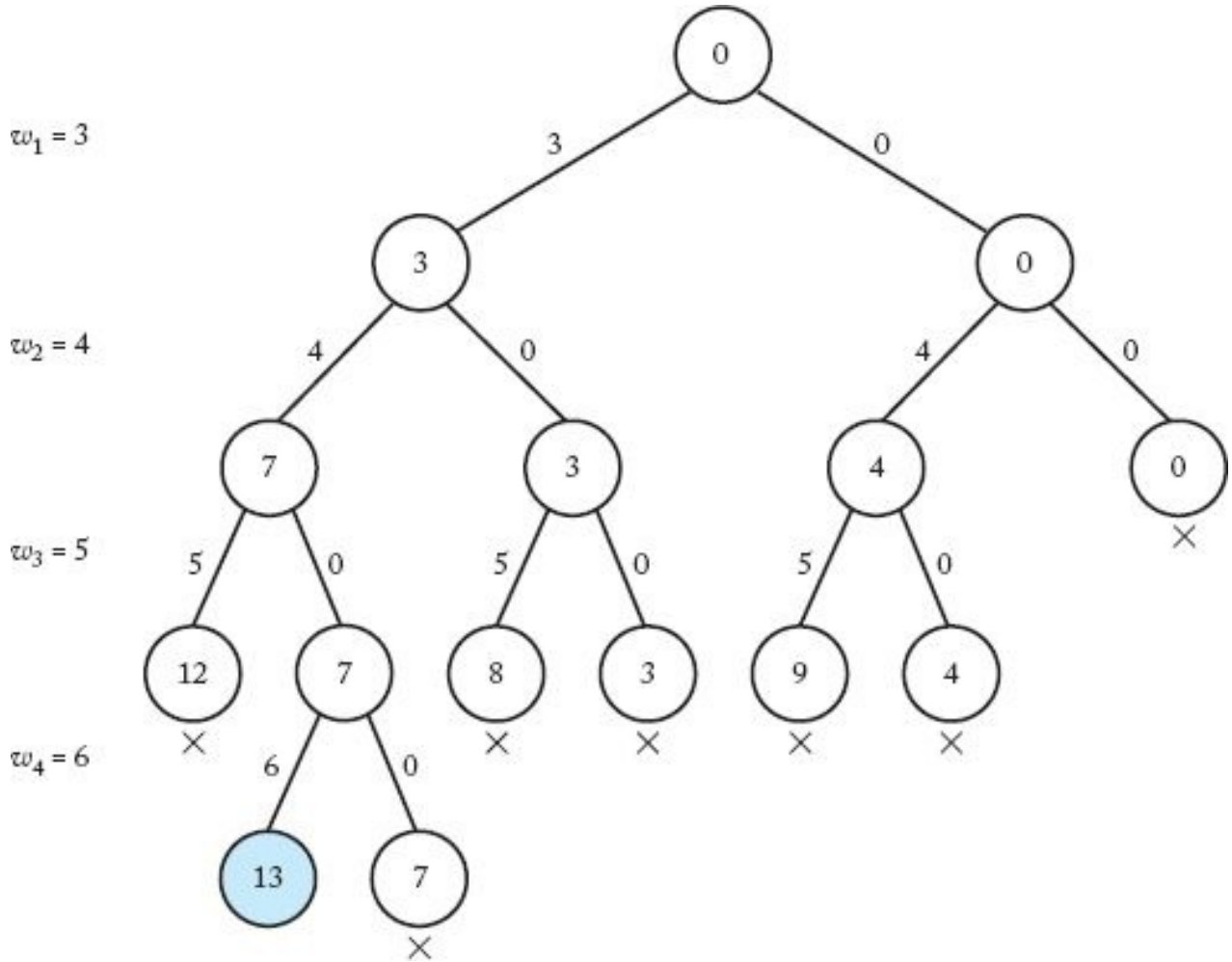
نشانهٔ دوم برای غیرامیدبخش بودن

If, at a given node, adding all the weights of the remaining items to weight does not make weight at least equal to W, then weight could never become equal to W by expanding beyond the node. This means that if $total$ is the total weight of the remaining weights, a node is nonpromising if

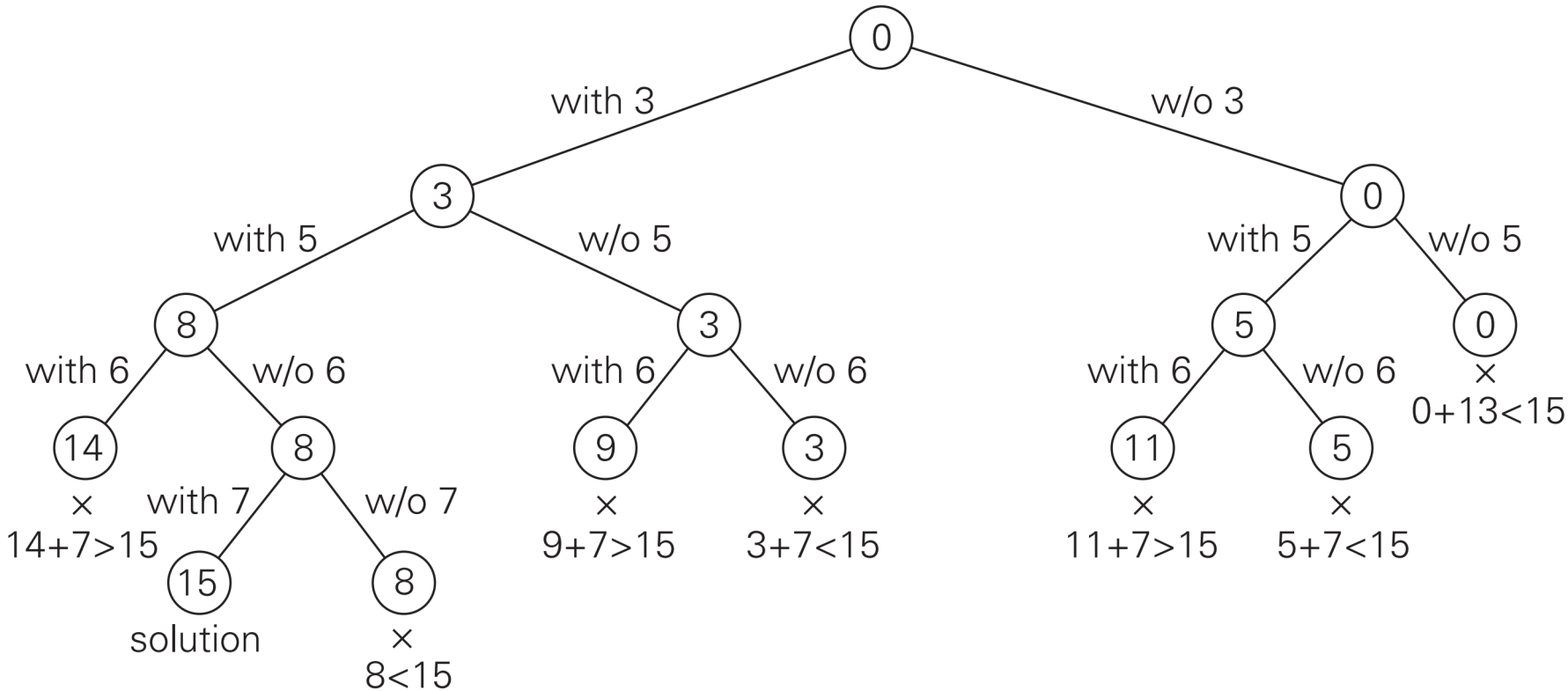
$$weight + total < W.$$

$weight + w_{i+1} > W$ ☞ **The sum $weight$ is too large.**
$weight + total < W$ ☞ **The sum $weight$ is too small.**

$$n = 4,\ W = 13,\ \{w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6\}$$



$$n = 4,\ W = 15,\ \{w_1 = 3, w_2 = 5, w_3 = 6, w_4 = 7\}$$



When the sum of the weights included up to a node equals W, there is a solution at that node. Therefore, we cannot get another solution by including more items. This means that if

$$W = weight,$$

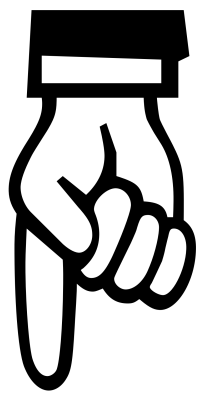
we should print the solution and backtrack.

Some backtracking algorithms sometimes find a solution before reaching a leaf in the state space tree. This is one such algorithm.

The following algorithm uses an array $include$. It sets $include[i]$ to "yes" if $w[i]$ is to be included and to "no" if it is not.

The Backtracking Algorithm for the Sum-of-Subsets Problem

Problem: Given n positive integers (weights) and a positive integer W, determine all combinations of the integers that sum to W.
Inputs: positive integer n, sorted (nondecreasing order) array of positive integers w indexed from 1 to n, and a positive integer W.
Outputs: all combinations of the integers that sum to W.



```
void sum_of_subsets (index i,
                     int weight, int total)
{
   if (promising(i))
      if (weight == W)
         cout << include[1] through include[i];
      else{
         include[i + 1] = "yes";                    // Include w[i + 1].
         sum_of_subsets(i + 1, weight + w[i +1], total - w[i + 1]);
         include[i + 1] = ''no";                    // Do not include w[i + 1].
         sum_of_subsets(i + 1, weight, total - w[i + 1]);
      }
}

bool promising (index i);
{
   return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}
```

Following our usual convention, n, w, W, ***and*** $include$ ***are not inputs to our routines. If these variables were defined globally, the top-level call to*** $sum\_of\_subsets$ ***would be as follows:***

$$sum\_of\_subsets(0, 0, \sum_{j=1}^{n} w[j]).$$

A leaf in the state space tree that does not contain a solution is nonpromising because there are no weights left that could bring the value of weight up to W***. This means that the algorithm should not need to check for the terminal condition*** $i = n$.

(توضیح بیشتر در اسلاید بعد)

چرا برگ‌ها خودبه‌خود غیرامیدبخش تشخیص داده می‌شوند؟

When $i = n$***, the value of total is*** 0 ***(because there are no weights remaining). Therefore, at this point***

$$weight + total = weight + 0 = weight,$$

which means that $weight + total \geq W$ ***is true only if*** $weight \geq W$***. Because we always keep*** $weight \leq W$***, we must have*** $weight = W$***. Therefore, when*** $i = n$***, function promising returns true only if*** $weight = W$***. But in this case there is no recursive call because we found a solution. Therefore, we do not need to check for the terminal condition*** $i = n$***.***

The number of nodes in the state space tree searched by our algorithm is equal to

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^n = 2^{n+1} - 1.$$

یک مثال مهم که در آن، الگوریتم ما مجبور به ملاقات تعدادی نمایی نود است:

Given only this result, the possibility exists that the worst case could be much better than this. That is, it could be that for every instance only a small portion of the state space tree is searched. This is not the case. For each n, it is possible to construct an instance for which the algorithm visits an exponentially large number of nodes. This is true even if we want only one solution. To this end, if we take

$$\sum_{i=1}^{n-1} w_i < W, \qquad w_n = W,$$

there is only one solution $\{w_n\}$, and it will not be found until an exponentially large number of nodes are visited.

Even though the worst case is exponential, the algorithm can be efficient for many large instances.

Even if we state the problem so as to require only one solution, the Sum-of-Subsets problem, like the 0-1 Knapsack problem, is intractable.

CNF-Satisfiability Problem (CNF-SAT)

A logical (boolean) variable is a variable that can have one of two values: true or false. If x is a logical variable, $\overline{x}$ is the negation of x. That is, x is true if and only $\overline{x}$ if is false. A literal is a logical variable or the negation of a logical variable. A clause is a sequence of literals separated by the logical or operator ($\vee$). A logical expression in conjunctive normal form (CNF) is a sequence of clauses separated by the logical and operator ($\wedge$). The following is an example of a logical expression in CNF:

$$(\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_4}) \wedge (\overline{x_2} \vee x_3 \vee x_4)$$

نقیض متغیر تصمیم دودویی x را با $\neg x$ نیز نشان می‌دهند.

The CNF-Satisfiability (CNF-SAT) Decision problem is to determine, for a given logical expression in CNF, whether there is some truth assignment (some set of assignments of true and false to the variables) that makes the expression true.

NP-completeness: No one has ever found a polynomial-time algorithm for this problem, and no one has ever proven that it cannot be solved in polynomial time.

For the instance

$$(x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2})$$

the answer to CNF-Satisfiability is "yes," because the assignments $x_1 = true$, $x_2 = false$, *and* $x_3 = false$ *make the expression true. For the instance*

$$(x_1 \vee x_2) \wedge (\overline{x_1}) \wedge (\overline{x_2})$$

the answer to CNF-Satisfiability is "no," because no assignment of truth values makes the expression true.

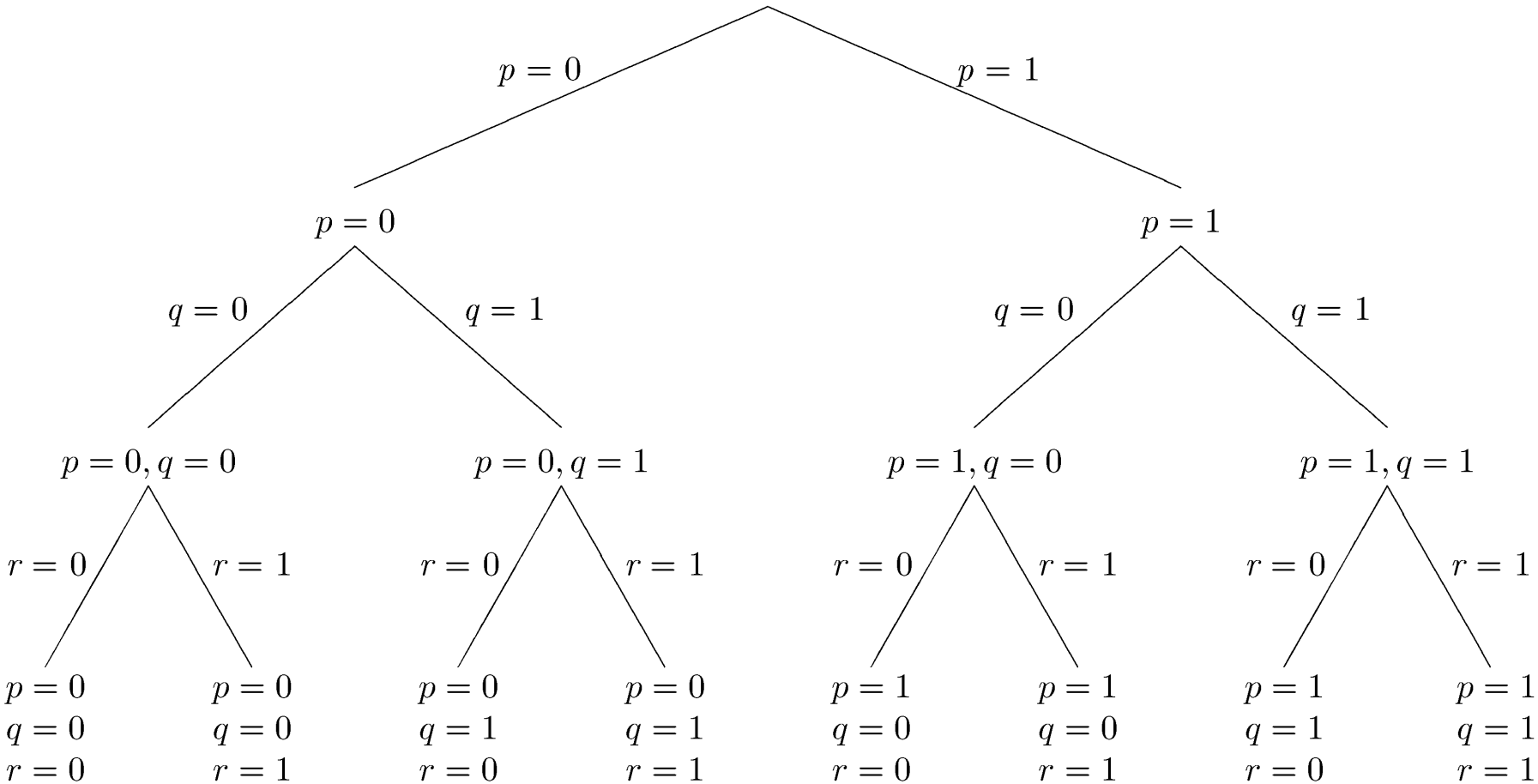
Truth Table Method

$$\Delta = (p \vee q) \wedge (p \vee \overline{q}) \wedge (\overline{p} \vee q) \wedge (\overline{p} \vee \overline{q} \vee \overline{r}) \wedge (\overline{p} \vee r)$$

p	q	r	$p \vee q$	$p \vee \neg q$	$\neg p \vee q$	$\neg p \vee \neg q \vee \neg r$	$\neg p \vee r$	Δ
0	0	0	0	1	1	1	0	0
0	0	1	0	1	1	1	1	0
0	1	0	1	0	1	1	1	0
0	1	1	1	0	1	1	1	0
1	0	0	1	1	0	1	0	0
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	0	1	0

There is one row for each possible truth assignment. For each truth assignment, each of the sentences in Δ is evaluated. If any sentence evaluates to 0, then Δ as a whole is not satisfied by the truth assignment. If a satisfying truth assignment is found, then Δ is determined to be satisfiable. If no satisfying truth assignment is found, then Δ is unsatisfiable. In this example, every row ends with Δ not satisfied. So the truth table method concludes that Δ is unsatisfiable.

State-Space Tree



Suppose, for example, a proposition p has been assigned the truth value 1. (1) Each disjunction (clause) containing a disjunct (literal) p may be ignored because it must already be satisfied by the current partial assignment. (2) Each disjunction φ containing a disjunct $\neg p$ may be modified (call the result φ') by removing from it all occurrences of the disjunct $\neg p$ because, under all truth assignments where p has truth value 1, φ holds if and only if φ' holds.

If a proposition p has been assigned the truth value 0, we can simplify our sentences analogously. (1) Each disjunction containing a disjunct $\neg p$ may be ignored because it must already be satisfied by the current partial assignment. (2) Each disjunction φ containing a disjunct p may be modified by removing from it all occurrences of the disjunct p.

در هر مرحله، یک متغیر که هنوز مقداری به آن اختصاص نیافته است را انتخاب کرده، مقدار آن برابر با 1 یا 0 قرار میدهیم. اگر مثلاً قرار دهیم $x = 1$ آنگاه همهٔ کلاوزهایی که حاوی x هستند از فرمول حذف میشوند. چرا؟ چون خیالمان راحت است که به ازای این مقداردهی، کلاوز مذکور قطعاً درست خواهد بود. اما از طرف دیگر، باید لیترال $\neg x$ را از همهٔ کلاوزها حذف کرد. چرا؟ چون مقدار این لیترال غلط است، و اگر قرار باشد که مقدار یک کلاوز که حاوی چنین لیترالی است درست باشد، باید بقیهٔ لیترال‌ها به‌شکل مناسبی مقداردهی شوند. حالا اگر قرار دهیم $x = 0$ چطور؟

Most SAT solvers are based on the following simple backtracking algorithm:

proc backtracking (F : clause set) : **bool**

// outputs 1, if F is satisfiable, 0 otherwise

if $\square \in F$ **then return** 0

if $F = \emptyset$ **then return** 1

choose a variable $x \in Var(F)$

if backtracking $(F\{x = 0\})$ **then return** 1

return backtracking $(F\{x = 1\})$

تذکر: منظور از کاراکتر $\square$، یک کلاوز تهی است، و منظور از کاراکتر $\emptyset$ آن است که دیگر فرمول F حاوی هیچ کلاوزی نیست (یعنی همهٔ کلاوزها به‌واسطهٔ ارضا شدن حذف شده‌اند).

In principle, the backtracking algorithm is more efficient than simply trying all assignments for F, since in case $\square \in F\alpha$ for some partial assignment α created in the recursive process, it follows that $F\beta = 0$ for all extensions β from α. In this case the algorithm does not need to visit the whole subtree with the extensions of α and the search space becomes smaller. The backtracking algorithm presented here can be seen as a "meta algorithm." It describes a class of algorithms because the step "choose a variable $x \in Var(F)$" is not completely specified. There can be many possibilities for the choice of the next variable x to be assigned.

$$\Delta = (p \vee q) \wedge (p \vee \overline{q}) \wedge (\overline{p} \vee q) \wedge (\overline{p} \vee \overline{q} \vee \overline{r}) \wedge (\overline{p} \vee r)$$

Under the partial assignment $p = 1$, we can simplify our sentences as shown below. The first two sentences are dropped, and the literal $\neg p$ is dropped from the other three sentences.

Original	Simplified
$p \vee q$	–
$p \vee \neg q$	–
$\neg p \vee q$	q
$\neg p \vee \neg q \vee \neg r$	$\neg q \vee \neg r$
$\neg p \vee r$	r