Exhaustive Search and Backtracking

Chapter 9

2016

Problem solving

- Backtracking is one of the most general techniques in algorithm design. It deals with finding an *optimal solution* from a set of solutions.

- Backtracking can be applied to those problems for which the solution is expressible as a tuple or vector $x = (x_1, x_2, ..., x_n) \in S_1 \times S_2 \times ... \times S_n = S$, where S_i are finite sets with m_i elements, for $1 \leq i \leq n$. The set S is called *solution space*.

Optimization Problem

- Often we are asked to find one or all the solutions that maximize (or minimize or satisfy) a *criterion function* $C(x_1,x_2,...,x_n)$.

- Exhaustive (brute-force) approach:
  - when all the solutions have the same length *n* one approach is to generate the Cartesian product $S_1 \times S_2 \times ... \times S_n$ and save only the solutions that satisfy the criterion function. This method is highly inefficient.

Exhaustive Search

- Let us assume that the exhaustive search space is a set $S = \{s_1, s_2, \ldots, s_N\}$ where N is the total number of states, usually a very large number.
- The elements of S can be generated using an iterative algorithm, in a specific ordering, according to the following scheme:
  - Use a *Gen* procedure that generates the next element.
  - *Gen* takes a Boolean parameter *gc* to control the generation process.
    - If *gc* = *false* then the generating process is reset, and the first value $s_1 \in S$ is generated
    - If *gc* = *true* then the next value $s \in S$ is generated
  - *Gen* returns also a new value for *gc*, as follows:
    - If *gc* = *true* then the currently generated *s* is not the last value of *S*.
    - If *gc* = *false* then the currently generated *s* is the last value of *S*, so the generating process was finished.

Cartesian Products

- Let $A_1, A_2, ..., A_m$ be m finite sets, $A_i = \{ 1, 2, ..., n_i \}$, $1 \leq i \leq m$. The problem is to generate all the $\Pi_{1 \leq i \leq m} n_i$ elements of the Cartesian product $P = A_1 \times A_2 \times ... \times A_m$.
- We present an algorithm for generating the elements of P in increasing lexicographical order.
- For example, let $m = 3$, $n_1 = 2$, $n_2 = 3$, $n_3 = 2$, $A_1 = \{ 1, 2 \}$, $A_2 = \{ 1, 2, 3 \}$, $A_3 = \{ 1, 2 \}$.
- The $2 \times 3 \times 2 = 12$ elements of $A_1 \times A_2 \times A_3$, generated in increasing lexicographical order are: [1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 2, 2], [1, 3, 1], [1, 3, 2], [2, 1, 1], [2, 1, 2], [2, 2, 1], [2, 2, 2], [2, 3, 1], [2, 3, 2].
- Observation: the vector v representing an element of P can be interpreted as a number written in a generalized basis $(n_1, n_2, \ldots, n_m)$. The digits are counted starting from 1. For example, if $n_1 = n_2 = \ldots = n_m = 2$ then we obtain a binary representation (with the convention that digit $1 \rightarrow 0$ and digit $2 \rightarrow 1$), as: [2, 1, 1] is the binary number 100_2.

Generating Method

- The first element to be generated is the vector [1, 1, ..., 1] with m elements.
- Let $[v_1, v_2, ..., v_m]$ be an element of P. To generate the successor of v, we examine v from right to left and determine the first i such that $v_i < n_i$. Then, the successor of v will be v', defined as $v' = [v_1, v_2, ..., v_{i-1}, v_i + 1, 1, ..., 1]$.
- For example, if $v = [1, 2, 2]$, then $i = 2$ and $v' = [1, 3, 1]$.
- If there is no i such that $v_i < n_i$, it means that $v_i = n_i$ for all $1 \leq i \leq m$ and the generating process is finished.
- This algorithm is implemented in procedure *CartProd* on the next slide.

Algorithm for Cartesian Products

- We assume that *n* is a vector with *m* elements such that *n*[*i*] represents the number of elements of the set A_i.
- The generating process can be implemented as *do-while* loop such that:
  - Initially (before the loop) *gc* is set to *false*.
  - In the loop, first call CART-PROD(*n*,*m*,*v*,*gc*)
  - After each call a new element *v* of the Cartesian product is generated.
  - The loop terminates when *gc* is again *false*.

```
CART-PROD($n, m, v, gc$)
1.  if $\neg gc$ then
2.      for $i = 1, m$ do
3.          $v[i] \leftarrow 1$
4.      $gc \leftarrow true$
5.      return
6.  else
7.      for $i = m, 1, -1$ do
8.          if $v[i] < n[i]$ then
9.              $v[i] \leftarrow v[i] + 1$
10.             return
11.         $v[i] \leftarrow 1$
12.     $gc \leftarrow false$
13.     return
```

Homework

1. Let A be a set with n elements. Develop a method and algorithm for generating all the subsets of set A. For example, if $n = 3$ then the subsets are: {}, {1}, {2}, {1,2}, {3}, {1,3}, {2,3}, {1,2,3}.
2. Let A be a set with n elements and k a natural number between 0 and n, i.e. $0 \leq k \leq n$. Develop a method and algorithm for generating all the subsets of set A with k elements. For example, if $n = 4$ and $k = 2$ then the subsets are: {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}.
3. Let A be a set with n elements. Develop a method and algorithm for generating all the permutations of set A. For example, if $n = 3$, the permutations are: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1.

Backtracking approach

- Backtracking avoids the exhaustive approach as follows:
  - the solution is built one element at a time, i.e. the component x_k will get a value from S_k if and only if $x_1, x_2, ..., x_{k-1}$ have already got values and the partial solution $(x_1, x_2, ..., x_k)$ has chances to be completed to a solution $(x_1, x_2, ..., x_n)$.
  - the partial vector $(x_1, x_2, ..., x_k)$ must satisfy some special conditions $B_k(x_1, x_2, ..., x_k)$ called *bounding functions*, in order for the partial solution to have chances of success. If these conditions are not satisfied then x_k will get another possible value from the set S_k, or if all the values from S_k were tried then k will be decremented by one and the search is continued.

Backtracking advantage

- If the partial vector $(x_1, x_2, ..., x_k)$ cannot be completed to a solution vector then $m_{k+1}\cdot...\cdot m_n$ potential solutions will be completely ignored by the search process.

Iterative backtracking

- Let $(x_1, x_2, ..., x_{k-1})$ be a partial solution.
- Let $T(x_1, x_2, ..., x_{k-1})$ be the set of all possible values for x_k (the *trial set* for x_k).
- Let $B_k(x_1, x_2, ..., x_k)$ be the bounding functions expressed as predicates i.e. $B_k(x_1, x_2, ..., x_k)$ is true if and only if the partial solution $(x_1, x_2, ..., x_k)$ has chances to be completed to a solution.
- Thus the candidates for position k of the solution x are the values in $x \in T(x_1, x_2, ..., x_{k-1})$ that satisfy $B_k(x_1, x_2, ..., x_k)$ for $x_k = x$.

Iterative backtracking algorithm

```
ITERATIVE-BACKTRACKING
1. $k \leftarrow 1$
2. **while** $k > 0$ **do**
3.         **if** there is an untried $x[k] \in T(x[1], \ldots, x[k-1])$
             such that $B_k(x[1], \ldots, x[k])$ is true **then**
4.             **if** $(x[1], \ldots, x[k])$ is a solution **then**
5.                 save solution $(x[1], \ldots, x[k])$
6.             $k \leftarrow k + 1$
7.         **else**
8.             $k \leftarrow k - 1$
```

A refinement of iterative backtracking

- Quite often the sets S_k are arithmetic progressions $\{a_k,a_k+h_k,...,b_k\}$ for all $1\leq k\leq n$ and all the solutions have the same length n.

```
REFINED-ITERATIVE-BACKTRACKING($n, a, b, h$)
1.  $k \leftarrow 1$
2.  $x[1] \leftarrow a[1] - h[1]$
3.  while $k > 0$ do
4.          $x[k] \leftarrow x[k] + h[k]$
5.          while $(x[k] \leq b[k]) \wedge \neg B_k(x[1], \ldots, x[k])$ do
6.                  $x[k] \leftarrow x[k] + h[k]$
7.          if $x[k] \leq b[k]$ then
8.              if $k = n$ then
9.                  save solution $(x[1], \ldots, x[n])$
10.             else
11.                 $k \leftarrow k + 1$
12.                 $x[k] \leftarrow a[k] - h[k]$
13.         else
14.             $k \leftarrow k - 1$
```

Recursive backtracking

```
RECURSIVE-BACKTRACKING($k$)
1. for every $x[k] \in T(x[1], \ldots, x[k-1])$
         such that $B_k(x[1], \ldots, x[k])$ is true do
2.       if $(x[1], \ldots, x[k])$ is a solution then
3.          save solution $(x[1], \ldots, x[k])$
4.       RECURSIVE-BACKTRACKING($k + 1$)
```

The procedure for recursive backtracking should be called as:

```
RECURSIVE-BACKTRACKING(1)
```

In order to limit the depth of the search we must enclose the body of this algorithm into an if statement that checks if $k \leq n$; n is the maximum depth of the search (length of a solution).

The *n*-queens problem

- Suppose it is given a $n \times n$ chessboard. The problem is to determine all possibilities to place n queens on the chessboard so that no two of them attack, i.e. no two of them are on the same row, column or diagonal.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

Solution (4,6,8,2,7,1,3,5) for $n = 8$

- Let us number the rows and columns of the chessboard 1 through n.
- Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i.
- A solution can be represented as a vector $(x_1, x_2, ..., x_n)$, where x_i is the column on which queen i is placed.
- The explicit constraints express the fact that $1 \leq x_i \leq n$, so $S_i = \{ 1, 2, ..., n \}$.
- The implicit constraints are that no two x_i's can be the same (i.e. all queens must be on different columns), and no two queens can be on the same diagonal, i.e. $x_i \neq x_j$ and $|i-j| \neq |x_i-x_j|$ for all $1 \leq i, j \leq n$, $i \neq j$.

Solving the *n*-queens problem (I)

```
#include <stdio.h>
#include <stdlib.h>
#define NMAX 50
#define TRUE  1
/* Bounding functions */
int bound(int);
/* Print a solution */
void print(int);
/* Compute solutions */
void queens(int);
/* Number of solutions */
int nsol;
/* Solution */
int x[NMAX];
```

```
int bound(int k) {
  int i,p=TRUE;
  for (i=1;i<k;i++)
    p = p && ((x[i]!=x[k]) &&
         (abs(i-k)!=abs(x[i]-x[k])));
  return p;
}

void print(int n) {
  int i;
  nsol++; printf("%4d.  ",nsol);
  for (i=1;i<=n;i++)
    printf("%4d",x[i]);
  printf("\n");
  return;
}
```

Solving the *n*-queens problem (II)

```
void queens(int n) {
  int k=1;
  x[1]=0;
  nsol = 0;
  while (k>0) {
    x[k]++;
    while ((x[k]<=n)
      && !bound(k))
      x[k]++;
    if (x[k]<=n)
      if (k==n)
        print(n);
      else
        x[++k]=0;
    else
      k--;
  }
}
```

```
void main() {
  int n;
  scanf("%d",&n);
  while (n>0) {
    printf("  n = %4d\n",n);
    printf("  No.        Solution\n");
    printf("---------------------\n");
    queens(n);
    scanf("%d",&n);
  }
}
```

Solving the *n*-queens problem (III)

Input:

```
5
6
0
```

Output:

```
n =    5
No.          Solution
 1.      1   3   5   2   4
 2.      1   4   2   5   3
 3.      2   4   1   3   5
 4.      2   5   3   1   4
 5.      3   1   4   2   5
 6.      3   5   2   4   1
 7.      4   1   3   5   2
 8.      4   2   5   3   1
 9.      5   2   4   1   3
10.      5   3   1   4   2
n =    6
No.          Solution
 1.      2   4   6   1   3   5
 2.      3   6   2   5   1   4
 3.      4   1   5   2   6   3
 4.      5   3   1   6   4   2
```

The discrete knapsack problem

- We are given a knapsack of capacity M>0 and n objects of weights w_i, $1 \leq i \leq n$. Adding object i to the knapsack gives a profit p_i>0. We must find a subset of objects to add to the knapsack that yields a maximum profit.
- A solution is a tuple $(x_1,\ldots,x_n) \in \{0,1\}^n$ such that $\Sigma_{1\leq i\leq n} w_i x_i \leq M$ and $\Sigma_{1\leq i\leq n} p_i x_i$ is maximum.
- We can use backtracking with $S_i = \{0,1\}$. Let p be the maximum value of the profit that has been found so far. Initially $p = 0$. We have two bounding functions:
  - One that expresses that the currently selected objects do not overweight the knapsack capacity M, i.e. if $(x_1,\ldots,x_k)$ is a partial solution then $\Sigma_{1\leq i\leq k} w_i x_i \leq M$.
  - One that expresses that the current partial solution can be extended to a solution that yields a higher profit that the current highest profit, i.e. $\Sigma_{1\leq i\leq k} p_i x_i + \Sigma_{k+1\leq i\leq n} x_i > p$.

Solving the knapsack problem (I)

- Let's use recursive backtracking. The main function of the program is `pack(k,p_t,w_t,p_r)`.
  - The call assumes that positions 1 to k–1 of solution x have been filled in and it tries to find a value for $x[k]$.
  - *p_t* and *w_t* are the current profit and the current weight of the partial solution ($x[1], \ldots, x[k-1]$) and *p_r* is the maximum profit that could be obtained from all the remaining objects k, k+1, ..., n.
  - Initially $k = 1$, *p_t* = 0, *w_t* = 0 and *p_r* is the maximum profit (sum of all profits).
  - When a new object k is added to the knapsack we update *p_t* and *w_t* accordingly and subtract $p[k]$ from *p_r*.
  - Whenever a new solution is found, if the profit is higher than the current highest profit p then p is updated and the solution is saved.

Solving the knapsack problem (II)

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#define NMAX 100
void read_input(void);
void write_sol(void);
void pack(int, float,
  float, float);
void pack_sol(void);
typedef struct pw {
  float p;
  float w;
} PW;
/* Number of items */
int n;
/* Profits and weights */
PW p_w[NMAX];
/* Knapsack capacity */
float M;
```

```
/* Potential solution */
int x[NMAX];
/* Optimal solution */
int x_max[NMAX];
/* Total profit */
float p;
/* Maximum profit */
float p_max;
void read_input(void) {
  int i;
  p_max = 0;
  for (i=1;i<=n;i++) {
    scanf("%f%f",&p_w[i].p,&p_w[i].w);
    p_max += p_w[i].p;
  }
  scanf("%f",&M);
}
```

Input:

n

$p_1\ w_1$

...

$p_n\ w_n$

M

Solving the knapsack problem (III)

```
void pack(int k,float p_t,
  float w_t,float p_r) {
  if (k<=n) {
    p_r = p_r-p_w[k].p;
    for (x[k]=0;x[k]<=1;x[k]++) {
      w_t += x[k]*p_w[k].w;
      p_t += x[k]*p_w[k].p;
      if ((w_t<=M)&&(p_r+p_t>p)) {
        if (k==n) {
          memcpy(&x_max[1],&x[1],
            sizeof(int)*n);
          p = p_t;
        }
        else
          pack(k+1,p_t,w_t,p_r);
      }
    }
  }
}
```

```
void print_sol(int y[]) {
  int i;
  for (i=1;i<=n;i++) {
    printf("%2d",y[i]);
  }
  printf("\nProfit: %6.0f\n",p);
}
void pack_sol() {
  p = 0;
  pack(1,0,0,p_max);
}
void main() {
  scanf("%d",&n);
  while (n>0) {
    read_input();
    pack_sol();
    print_sol(x_max);
    scanf("%d",&n);
  }
}
```

Backtracking and graph searching

- The search process associated with a backtracking algorithm generates a *search tree* built according to the following rules:
  - Every tree node of level k (root has level 0) is labeled with a partial solution $(x_1,x_2,...,x_k)$; the root is labeled with the empty partial solution ();
  - From every node labeled $(x_1,x_2,...,x_{k-1})$ leaves one edge labeled with a value $x_k \in T(x_1,x_2,...,x_{k-1})$ to the child nodes labeled with $(x_1,x_2,...,x_k)$ such that the bounding functions are true;
- A backtracking algorithm performs a systematic exploration of the nodes of the the search tree in depth first order.

Example

- The next slide shows the search tree generated by the backtracking algorithm for solving the discrete knapsack problem for the following input data: n = 4, M = 10, p = (1,6,4,5), w = (1,7,2,2).

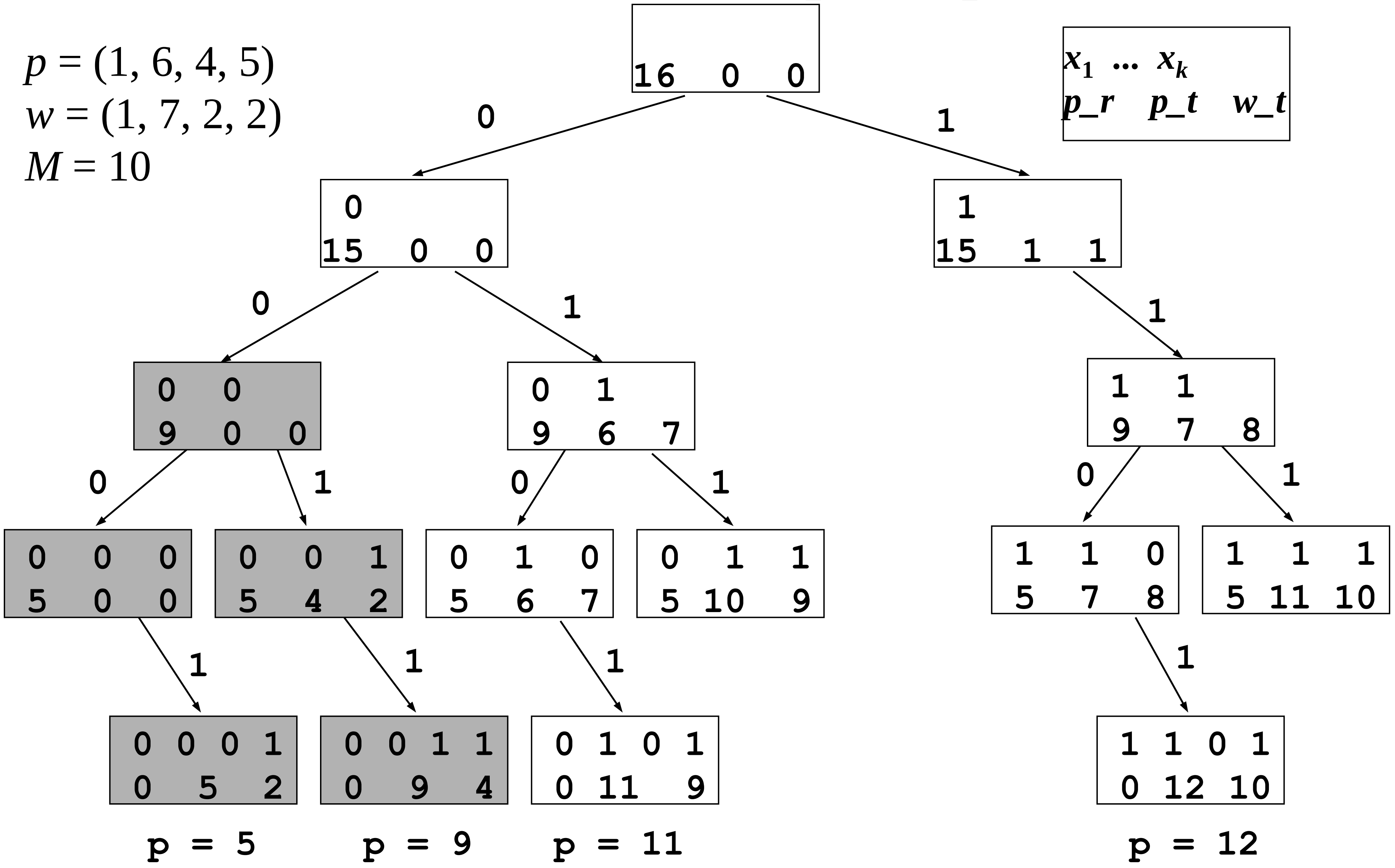
- Node values represent:
  - remaining max profit that can be achieved
  - current profit obtained
  - current weight.

- The optimal solution is: (1,1,0,1) that indicates the subset {1,2,4} of objects to be added to the knapsack with the profit 12.

- Without the use of the bounding functions, the search tree would have 2^{n+1}-1 nodes. For n = 4 this gives a total of 31 nodes. Note that using the bounding functions 15 nodes are pruned from the search tree.

Search tree for the knapsack problem

$p = (1, 6, 4, 5)$
$w = (1, 7, 2, 2)$
$M = 10$



Some comments

- Note that the search starts with a bad estimation of the maximal profit, i.e. with $p = 0$. This value is improved during the search: $p = 5$, $p = 9$, $p = 11$ and finally $p = 12$. The value of p is important for the bounding functions. A higher value of p means that the bounding functions will evaluate to false more often.
- The <u>search can be improved</u> if we start with a better estimate of the maximal profit. A good estimation is obtained if we compute a suboptimal solution using the strategy for solving the continuous knapsack problem, i.e. sort the objects in non-increasing order of the ratio profit/weight and add them to the knapsack in this order such that the knapsack capacity is not overweighed.
- With this strategy we obtain an initial estimation of 10 for the profit, corresponding to the solution {1,3,4}. Using this estimation, the search tree will have only 11 nodes, so 5 more nodes are pruned (the nodes grayed on the search tree).

Application: refined knapsack problem

- We can imagine that the value M represents an amount of money of an investment agent (the capital).
- The agent may choose to invest the capital in n offers.
- For each offer we have a capital w_i that should be invested and an estimated profit p_i.
- The knapsack problem asks for finding an optimal investment that maximizes the estimated profit.
- The problem can be refined by allowing the agent to buy a number of actions for each offer. For each offer i there are available n_i actions of value v_i. The refined problem can be solved using backtracking as well (left as homework!).

Efficiency of backtracking

- Even if backtracking avoids the exhaustive exploration of the solution space by pruning the solution tree with the help of the bounding functions, the resulting algorithm is most of the time exponential.
- The method is recommended when it is difficult or even impossible to devise an efficient polynomial algorithm for the problem in hand.
- The efficiency of backtracking algorithms depends on the following:
  - The time to generate the next value for x_k.
  - The number of elements of the sets S_k.
  - The time for evaluating the bounding functions.
  - The number of values for x_k that satisfy the bounding functions.
- There is a trade off in that bounding functions that are good also take more time to evaluate.
- The fourth factor determines the number of nodes generated and is dependent on the problem instance. The first three factors are relatively independent of the problem instance being solved. It is difficult to predict the number of nodes generated for a given problem instance.

Homework I

1. Se considera un graf $G = \langle V,E \rangle$ astfel incat $|V| = n$ si un set de m culori $\{1, 2, \ldots, m\}$. Se cere sa se determine toate colorarile posibile ale varfurilor grafului cu cate o culoare din cele m astfel incat orice doua varfuri vecine sa fie colorate diferit.

2. Se considera o multime A cu n elemente, $A = \{1, 2, \ldots, n\}$, astfel incat fiecare element i al lui A are o valoare v_i. Fie M un numar real dat. Sa se determine toate submultimile lui A care au suma valorilor elementelor egala cu M.

Homework II

3. Se considera un labirint de forma dreptunghiulara cu *m* linii si *n* coloane (numere naturale nenule). Anumite locatii din labirint sunt blocate (marcaj 1), iar altele sunt libere (marcaj 0). Se considera ca labirintul are un punct de intrare si un punct de iesire. Acestea sunt singurele locatii libere de pe marginea labirintului. Sa se determine daca exista un drum de la punctul de intrare la punctul de iesire din labirint. Se presupune ca miscarile in labirint au loc cu o pozitie pe orizontala sau pe verticala, cu conditia ca locatia destinatie sa fie libera. Se considera urmatorul labirint exemplu cu *m* = 9 linii si *n* = 8 coloane.