A Comparison of Exact and Heuristic Approaches to Capital Budgeting

Jindřiška Šedová, and Miloš Šeda

***Abstract*—**This paper summarizes and compares approaches to solving the knapsack problem and its known application in capital budgeting. The first approach uses deterministic methods and can be applied to small-size tasks with a single constraint. We can also apply commercial software systems such as the GAMS modelling system. However, because of NP-completeness of the problem, more complex problem instances must be solved by means of heuristic techniques to achieve an approximation of the exact solution in a reasonable amount of time. We show the problem representation and parameter settings for a genetic algorithm framework.

***Keywords*—**Capital budgeting, knapsack problem, GAMS, heuristic method, genetic algorithm.

I. Introduction

IN practice, we must often solve a problem of the following type: Suppose we wish to invest \$200. We have identified seven investment opportunities. Investment 1 requires \$40 and has a present value (a time-discounted value) of \$10; investment 2 requires \$50 and has a value of 15; and so on as shown in Table I. The question is: *Into which investments should we place our money so as to maximise our total present value*?

TABLE I
INPUT DATA

investment	1	2	3	4	5	6	7
requirement	40	50	20	60	40	70	40
present value	10	15	3	16	11	20	9

Let us denote by C the sum in question (it represents a constraint), by w_i the investment requirements, and by v_i the current values of these investments. If we consider a general case of n investments, then, formally written, we solve the following problem:

Manuscript received August 31, 2008. This work was supported in part by the Ministry of Education, Youth and Sports of the Czech Republic under research plan MSM 0021630518 "Simulation Modelling of Mechatronic Systems".

Jindřiška Šedová works in the Department of Law, Faculty of Economics and Administration, Masaryk University of Brno, Lipová 41a, 657 90 Brno, Czech Republic (phone: +420-5-49497663, e-mail: jsedova@econ.muni.cz).

Miloš Šeda works in the Institute of Automation and Computer Science, Faculty of Mechanical Engineering, Brno University of Technology, Technická 2896/2, CZ 616 69 Brno, Czech Republic (phone: +420-54114 3332; fax: +420-54114 2330; e-mail: seda@fme.vutbr.cz).

Maximise

$$\sum_{i=1}^{n} v_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \le C ,$$

where $x_i \in \{0,1\}$, $i = 1,2, \dots , n$.

The binary decision variables x_i specify whether or not the investment i is realized.

Evidently, the formulated *capital budgeting problem* is one of the applications of the well-known combinatorial optimisation problem, the 0-1 *knapsack problem* where the sum of money corresponds to the knapsack capacity, investment represent the items to be packed into the knapsack, their requirements relate to the items capacities (weights) and the current investment values to the item values (prices).

In practice, we may face a choice among projects that require investments of different amounts in each of several periods (with possibly different budgets C_j available in each periods) [17], with the return being realized over the life of the project. In this case, assuming n projects and m periods, we can model the problem as follows:

Maximise

$$\sum_{i=1}^{n} v_i\ x_i \tag{1}$$

subject to

$$\sum_{i=1}^{n} w_i x_i \le C_j ,\ j = 1,\dots,m , \tag{2}$$

where $x_i \in \{0,1\}$, $i = 1,2, \dots , n$ and

$$x_i = \begin{cases} 1, & \text{if we invest in project } i \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

This task corresponds to the m-dimensional knapsack problem (*Multi-Knapsack Problem*).

Both of these knapsack problems are NP-hard [7] and may only be solved for instances with a low number of inputs.

The knapsack problem and capital budgeting have numerous applications. In addition to capital markets, it may also be employed in areas such as information technologies [4], resource-constrained project scheduling [16], auditing [10] and health care [6].

In [8] and [12], fuzzy simulation-based genetic algorithms for solving capital budgeting problem in an uncertain economic environment are proposed.

In the following sections, we will restrict our considerations to the deterministic case and show possible ways of solving it with respect to its size.

II. Deterministic Methods

There are many deterministic techniques to solve the problem under investigation. These include:

- *Rude-force* approach is the simplest way based on generating all possible choices of investments and determining the optimal solution among them. This strategy is obviously not efficient, as its time complexity is $O(2^n)$.
- *Dynamic programming* approach is based on Bellman's principle of optimality. It can be used in a special case of the requirements and the investment capacity (sum of money in our example) being integers.
- *Branch-and-bound method* is the most efficient tool among these deterministic methods as it is based on a restriction of the search tree growth. Avoiding much enumeration depends on the precise upper bounds (the lower the upper bounds, the faster the finding of the solution is).

Let us solve the Knapsack Problem with capacity C=15 for the data from Table II.

TABLE II
Knapsack Problem with Ten Items

i	1	2	3	4	5	6	7	8	9	10
v_i	18	20	17	19	25	21	27	23	25	24
w_i	1	3	7	4	8	9	6	10	2	5

Let us denote $f_k(y)$ as maximal possible value using only first k items when the capacity limit is y.

$$f_i(y) = \max\left\{ \sum_{i=1}^{n} v_i x_i \,\middle|\, \sum_{i=1}^{n} w_i x_i \le y,\ x_i \in \{0,1\}, \right. \qquad (4)$$
$$\left. i = 1,\dots,k,\ 0 \le k \le n,\ 0 \le y \le C \right\}$$

Evidently, the maximal value of a knapsack with capacity 0 is 0, and the maximal value of any knapsack is 0 if we have no objects to put in it. That means:

$$f_k(0) = 0,\ \ 0 \le k \le n \qquad (5)$$

$$f_0(y) = 0,\ \ 0 \le y \le C \qquad (6)$$

General case $f_k(y)$:

a') If the k-th item *is not* a part of the maximal value of a knapsack, then $f_k(y) = f_{k-1}(y)$.

b') If the k-th item *is* a part of the maximal value of a knapsack, then $f_k(y) = f_{k-1}(y - w_k) + v_k$)

$$\Rightarrow\ \ f_k(y) = \max\{f_{k-1}(y),\ f_{k-1}(y - w_k) + v_k)\} \qquad (7)$$

Since it is not possible to put into the knapsack the item of weight w_k that would exceed the remaining capacity y, (7) implies $f_k(y) = f_{k-1}(y)$ for $y < w_k$.

Summarising all the cases, we get:

$$f_k(y) = \begin{cases} \max\{f_{k-1}(y),\ f_{k-1}(y - w_k) + v_k)\}, & \text{if } y \ge w_k \\ f_{k-1}(y), & \text{if } y < w_k \end{cases} \qquad (8)$$

If we increase the capacity of the knapsack and the set of objects that can be used to fill the knapsack and when y=C and k=n, then $F_n(C)$ corresponds to the maximum value of a knapsack with the full capacity, using all of the objects.

This algorithm can easily be implemented without recursive calls and thus there are no requirements for dynamic memory allocation when the program is running.

Computations by formulas (5), (6) and (8) may be expressed by the following statements:

```
for $k := 0$ to $n$ do
    $f[k,0] := 0;$
for $y := 0$ to $C$ do
    $f[0,y] := 0;$

for $k := 1$ to $n$ do
    for $y := 0$ to $C$ do
        if $y < w[k]$
          then $f[k,y] := f[k-1,y]$
          else if $f[k-1,y] > f[k-1,y-w[k]] + v[k]$
                  then $f[k,y] := f[k-1,y];$
                  else $f[k,y] := f[k-1,y-w[k]] + v[k];$
```

Trace-back in DP approach **(determination which items were put into the knapsack)**

0 We begin with the value $f_n(C)$, which, in matrix F with rows 0,1, ... , n and columns 0,1, ... , C is in the bottom right position.

1 This value is compared with the value which is in the previous row and in the same column. There are two possibilities:

2a Compared values are equal, then the last item (given by the row index of the investigated element in matrix F) is not included into the knapsack, it derives from a') and $f_k(y) = f_{k-1}(y)$ for $y < w_k$ in b')

2b Compared values differ, then we put the last item into the knapsack determining the remaining capacity and examine the value in the previous row and in the column that corresponds to the remaining capacity. If row 0 was reached, we finish, otherwise skip to Step 1.

For this procedure, it is true that none of the items in the trace back step are duplicated, i.e., each item may be included in the knapsack at most once. The trace-back can be expressed as follows:

```
for $i := 1$ to $n$ do
    $x[i] := 0;$
$k := n; y := C;$
while $k > 0$ do
    begin if $f[k,y] <> f[k-1,y]$
              then begin $x[k] := 1; y := y-w[k]$
                   end
              else $k := k-1$
    end;
```

From the previous explanations, the following assertion may be derived.

Theorem 1 *The previous algorithm for integer capacity C runs in O(nC) time.*

Proof. Time complexity is given by the dominant part represented by two nested cycles to evaluate matrix $f[0..n,0..C]$, the time complexity of the trace-back step is only $O(n)$. ■

The main drawback of the dynamic programming approach for solving the knapsack problem is that the time complexity depends not only on the number of items but also on the capacity C of the knapsack. In the next paragraph, we will show that this dependence does not occur in solving the problem by a branch-and-bound method.

Next disadvantage of the DP approach is that the number of columns in matrix F for $f_k(y)$ values and the corresponding allocation of memory increases out of proportion for a high value of integer capacity C. If C is not integer, then the DP approach even cannot be applied at all.

TABLE III
DYNAMIC PROGRAMMING COMPUTATION

$k\backslash y$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
2	0	18	18	20	38	38	38	38	38	38	38	38	38	38	38	38
3	0	18	18	20	38	38	38	38	38	38	38	55	55	55	55	55
4	0	18	18	20	38	38	38	39	57	57	57	57	57	57	57	74
5	0	18	18	20	38	38	38	39	57	57	57	57	63	63	63	74
6	0	18	18	20	38	38	38	39	57	57	57	57	63	63	63	74
7	0	18	18	20	38	38	38	45	57	57	65	65	65	55	84	84
8	0	18	18	20	38	38	38	45	57	57	65	65	65	55	84	84
9	0	18	25	43	43	45	63	63	63	70	82	82	90	90	90	91
10	0	18	25	43	43	45	63	63	67	70	82	87	90	90	94	106

As to time complexity of the algorithm, it is evident that, for the computation of a new matrix line, we only need to know the values of the previous row. Therefore, it is possible to work only with a two-row matrix. At the end of the computation of a new line, we mark its values as old and compute from them the next values. By these strategy, we save the memory size for $(n-1)*(C+1)$ matrix elements. On the other hand, we lose time by moving the values from the new to the old results. However, we can avoid this easily, by defining a flag in the subroutine which will keep a track of where the old solution is currently located and where the newly computed values are stored. Of course, when using this strategy, we must adapt the procedure so that it can save the information on which items are put into the knapsack.

If we apply the DP approach to the data from Table II and capacity C=15, then, from formulas (5), (6) and (8), we will get the following matrix $f[0..n,0..C]$ (see Table III) and we will use the trace-back step (marked by arrows) for determining the solution.

For the knapsack, items 10, 9, 4, 2 and 1 are selected and their total values is 24+25+19+20+18 = 106.

Finding the solution may be faster when we use the branch-and-bound method [11], [16], which restricts the growth of the search tree. Depending on the precision of the upper bounds, much enumeration may be avoided (the lower the upper bounds, the faster the finding of the solution). Let the items be numbered so that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}, \tag{9}$$

i.e. the unit values of the items form a nonincreasing sequence. Using *QuickSort, HeapSort,* or *MergeSort*, time complexity of this step is $O(n \log_2 n)$.

After this modification, we get the data as shown by Table IV.

TABLE IV
REORDERED DATA FOR BRANCH AND BOUND METHOD

i	1	2	3	4	5	6	7	8	9	10
v_i	18	25	20	24	19	27	25	17	21	23
w_i	1	2	3	5	4	6	8	7	9	10
v_i/w_i	18	12.5	6.66	4.8	4.75	4.5	3.12	2.42	2.33	2.3

We place items in the knapsack along this non-increasing sequence. Let $x_1, x_2, \dots, x_p$ be fixed values of 0 or 1 and

$$M_k = \left\{ \mathbf{x} \mid \mathbf{x} \in M,\, x_j = \xi_j,\, \xi_j \in \{0,1\},\, j = 1,\dots,p \right\} \tag{10}$$

where M is a set of feasible solutions. If

$$(\exists q)(p < q \leq n):\ \sum_{j=p+1}^{q-1} w_j \leq C - \sum_{j=1}^{p} w_j \xi_j < \sum_{j=p+1}^{q} w_j \tag{11}$$

then the upper bound for M_k can be determined as follows:

$$U_{\mathrm{B}}(M_k) = \sum_{j=1}^{p} v_j \xi_j + \sum_{j=p+1}^{q-1} v_j + \frac{v_q}{w_q}\left(C - \sum_{j=1}^{p} w_j \xi_j + \sum_{j=p+1}^{q-1} w_j \right) \tag{12}$$

The computation for the input data from Table IV is shown in Fig. 1.

For the solution of the task with multiple constraints, we must generalise the approaches mentioned above.

The combinatorial approach can be applied without any changes, but using the branch-and-bound method, we must redefine the upper bound. For its evaluation, we use the following formula

$$U_B(M_k) = \min\left\{ U_B^1(M_k), \dots, U_B^m(M_k) \right\} \tag{13}$$

where the auxiliary bounds $U_B^i(M_k),\, i = 1,\dots,m$ correspond to the given constraints and are determined as in the 0-1 knapsack problem. Before the evaluation of these auxiliary bounds, the other variables must be sorted again in the descending order of the values v_j/w_{ij}. Evidently, the run time will increase substantially.

Solving multiperiod capital budgeting by dynamic programming is quite inefficient and, therefore, we will not deal with it.

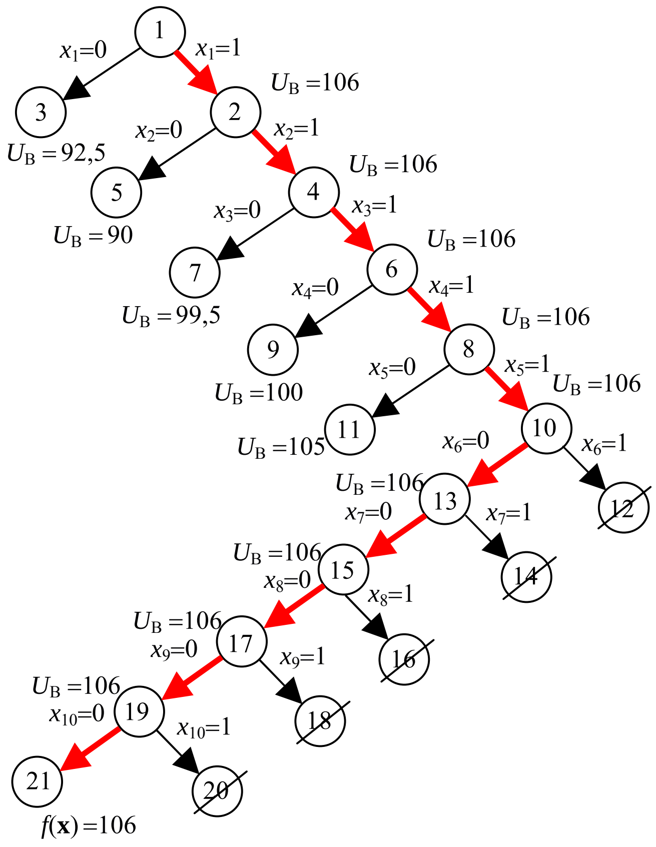

Fig. 1 Knapsack problem and its solution using a branch-and-bound method

III. Solving Capital Budgeting Problem Using Commercial Systems

For small instances of the problem, we can use the solver nested in Microsoft Excel, which can be applied to simple tasks of linear and integer programming. However, it fails for tasks with a higher number of inputs where, to a certain extent, more sophisticated software tools may be used such GAMS, LINDO, and LINGO.

The impetus for the development of GAMS, (*General Algebraic Modelling System*) [3], arose out of the frustrating experiences of an economic modelling group at the *World Bank*.

We illustrate GAMS for the input data from row W(I). For lack of space, we only mention that GAMS statements and objects are written here in capitals, comments are in small-case letters.

It can be seen that the programme is very close to the mathematical model.

```
$OFFSYMXREF
$OFFUELXREF
* section defining index sets
SETS
 I index  /1*7/;
* input data section
PARAMETERS
  W(I) requirements
     /1 400, 2 500, 3 200, 4 600, 5 400, 6 700, 7 400/
  Z(I) present values
      /1 100, 2 150, 3 30, 4 160, 5 110, 6 200, 7 90/;
SCALAR C investment limit (sum of money) /2000/;
*variable section (decision variables and objective function)
VARIABLES
  X(I) decision variable specifying whether or not
  *to realize investment i.
  CN total requirements
  CZ objective function (total present value);
BINARY VARIABLE X;
*equation section
EQUATIONS
  CONSTRAINT  budget constraints
  TOTREQ total requirements
  TOTPRESENT objective function (total present value);
  CONSTRAINT .. SUM(I,W(I)*X(I)) =L= C;
  TOTREQ .. SUM(I,W(I)*X(I)) =E= CN;
  TOTPRESENT .. SUM(I,Z(I)*X(I)) =E= CZ;
*choosing a model, running the solver, and
*displaying the results
MODEL INVEST /ALL/;
SOLVE INVEST USING MIP MAXIMIZING CZ;
DISPLAY CZ.L, CW.L, X.L;
```

However, GAMS and Excel are not effective in solving very complex multiperiod capital budgeting problems. In this case, we choose heuristic techniques.

IV. Heuristics

Heuristic [15] is a technique which seeks goal (i.e. near optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases, to state how close to optimality a particular feasible solution is.

Nowadays many heuristic approaches are used such as *repeated local search* [2], *simulated annealing* [2], *genetic algorithms* [13], [14], *tabu-search* [9] and *neural networks*. All these approaches are reviewed in [15].

Let us deal with the genetic algorithms now.

A GA program framework can be as follows:

```
generate an initial population;
evaluate fitness of individuals in the population;
repeat select parents from the population;
         recombine parents to produce children;
         evaluate fitness of the children;
         replace some or all of the population by the children
until a satisfactory solution has been found;
```

In the following paragraphs we briefly summarize GA settings to our scheduling problem.

Individuals in the population (*chromosomes*) are represented as binary strings of length *n*, where a value of 0 or 1 at the *i*-th bit (*gene*) implies that x_i=0 or 1 in the solution, respectively.

The *population size N* is chosen by Alander's experimental work [1] between *n* and 2*n*.

An *initial population* consists of *N* feasible solutions and it is obtained by generating random strings of 0's and 1's in the following way: First, all bits in all strings are set to 0, and then, for each of the strings, randomly selected bits are set to 1 until the solutions (represented by strings) are feasible.

The *fitness function* corresponds to the objective function to be maximised:

$$f(\mathbf{x}) = \sum_{i=1}^{n} v_i x_i \tag{14}$$

Pairs of chromosomes (parents) are selected for recombination by the *binary tournament selection* method, which selects a parent by randomly choosing 2 individuals from the population and selecting the most fit one.

The *recombination* is provided by the *uniform crossover* operator. That means that each gene in the child solution is created by copying the corresponding gene from one or the other parent, chosen by a binary random number generator. If the random number is 0, the gene is copied from the first parent; if it is 1, the gene is copied from the second parent. After crossover, the *mutation* operation is applied to each child. It works by inverting each bit in the solution with a small probability. We use a mutation rate of 5/*n* as a lower bound on the optimal mutation rate. It is equivalent to mutating five randomly chosen bits per string.

If we perform the crossover or mutation operations as described above, then the generated children can violate certain capacity constraints. We can assign *penalties* to these children that prevent infeasible individuals from entering the population.

A more constructive approach uses a *repair operator* that modifies the structure of an infeasible individual so that the solution becomes feasible. Its pseudo-Pascal code for the multi-knapsack problem is shown by the following algorithm. We assume that variables are sorted and renumbered in the descending order of their pseudo-utility ratios $u_i = v_i/w_{ji}$ [5].

```
$J := \{1, \ldots, m\}$
$W_j := \sum_{i=1}^{n} w_{ji} x_i,\ \forall j \in J;$
  for $i := n$ downto 1 do
    if $(x_i = 1)$ and $(W_j > C_j;$ for any $j \in J)$
      then begin $x_i := 0;$
                 $W_j := W_j - w_{ji},\ \forall j \in J$
           end;
```

Once a new feasible child solution has been generated, the child will replace a randomly chosen solution. We use a *steady-state* replacement technique based on eliminating the individual with the lowest fitness value.

Since the optimal solution values for most problems are not known, the termination of a GA is usually controlled by specifying a maximum number of generations $t_{\max}$. We choose $t_{\max} \leq 5000$.

V. Conclusion

In the previous paragraphs, several approaches to capital budgeting, based on solving knapsack problem, were discussed. It was shown that, for small instances, deterministic methods such as the branch-and-bound method and, in a special case, the dynamic programming approach may also be used to obtain better results than heuristic methods. On the contrary, the branch-and-bound method and dynamic-programming approach are not efficient in multiperiod capital budgeting with many periods and investments because of the exponentially growing time in branch-and-bound method calculations and high memory requirements for static arrays in the dynamic-programming approach.

In these cases, we choose heuristic techniques. The genetic algorithm has been found to work efficiently. It provides good results in a reasonable amount of time for hundreds of items and tens of capacity constraints.

In the future, we will implement other heuristics and consider uncertain economic influences.

References

[1] J. T. Alander, "On Optimal Population Size of Genetic Algorithms," in *Proceedings of CompEuro 92*, IEEE Computer Society Press, pp. 65-70, 1992.

[2] R. Battiti and G. Tecchioli, "Local Search with Memory: Benchmarking RTS," *Operations Research Spectrum*, vol. 17, no. 2/3, pp. 67-86, 1996.

[3] A. Brooke, D. Kendrick and A. Meeraus, *GAMS, release 2.25. A User's Guide*. Danvers, Massachussetts: Boyd & Fraser Publishing Company, 1992.

[4] A. Chandra, N.M. Menon and B.K. Mishra, "Budgeting for Information Technology," *International Journal of Accounting Information Systems*, vol. 8, issue 4, pp. 264-282, 2007.

[5] P. Chu, "A Genetic Algorithm Approach for Combinatorial Optimisation Problems," PhD thesis, The Management School Imperial College of Science, Technology and Medicine, London, 1997.

[6] L. Eldenburg and R. Krishnan, "Management Accounting and Control in Health Care: An Economics Perspective," *Handbooks of Management Accounting Research*, vol. 2, pp. 859-883, 2006.

[7] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Company, 1997 (19th printing).

[8] X. Huang, "Mean-variance Model for Fuzzy Capital Budgeting," *Computers & Industrial Engineering*, vol. 55, issue 1, pp. 34-47, 2008.

[9] F. Glover and M. Laguna, *Tabu Search*. Boston: Kluwer Academic Publishers, 1997.

[10] D. Kim, "Capital Budgeting for New Projects: On the Role of Auditing in Information Acquisition," *Journal of Accounting and Economics*, vol. 41, issue 3, pp. 257-270, 2006.

[11] J. Klapka, J. Dvořák and P. Popela. *Methods of Operational Research* (in Czech). Brno: VUTIUM, 2001.

[12] R. Liang and J. Gao, "Dependent-Chance Programming Models for Capital Budgeting in Fuzzy Environments," *Tsinghua Science & Technology*, vol. 13, issue 1, pp. 117-120, 2008.

[13] Z. Michalewicz and D.B. Fogel, *How to Solve It: Modern Heuristics*. Berlin: Springer-Verlag, 2000.

[14] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer Verlag, 1996.

[15] C.R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems*. Oxford: Blackwell Scientific Publications, 1993.

[16] M. Šeda, "Solving Resource-Constrained Project Scheduling Problem As a Sequence of Multi-Knapsack Problems," *WSEAS Transactions on Information Science and Applications*, vol. 3, issue 10, pp. 1785-1791, 2006.

[17] M.A. Trick, "A Tutorial on Integer Programming", http://mat.gsia.cmu.edu/orclass/integer/integer.html, 1997.