Computer tools in education, 2022
№ 2: 41–58
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-2-41-58

ON THE CLASSICAL VERSION OF THE BRANCH AND BOUND METHOD

Melnikov B. F.[1,2], PhD, Professor, ✉ bormel@mail.ru, orcid.org/0000-0002-6765-6800
Melnikova E. A.[2], PhD, Associated Professor, ya.e.melnikova@yandex.ru, orcid.org/0000-0003-1997-1846

[1]Shenzhen MSU – BIT University, No. 1, International University Park Road, Dayun New Town, Longgang District, Shenzhen, PRC, 517182, Guangdong Province, Shenzhen, China
[2]Russian State Social University, 4, build. 1, Wilhelm Pieck street, 129226, Moscow, Russia

Abstract

In the computer literature, a lot of problems are described that can be called discrete optimization problems: from encrypting information on the Internet (including creating programs for digital cryptocurrencies) before searching for "interests" groups in social networks. Often, these problems are very difficult to solve on a computer, hence they are called "intractable". More precisely, the possible approaches to quickly solving these problems are difficult to solve (to describe algorithms, to program); the brute force solution, as a rule, is programmed simply, but the corresponding program works much slower.

Almost every one of these intractable problems can be called a mathematical model. At the same time, both the model itself and the algorithms designed to solve it are often created for one subject area, but they can also be used in many other areas. An example of such a model is the traveling salesman problem. The peculiarity of the problem is that, given the relative simplicity of its formulation, finding the optimal solution (the optimal route). This problem is very difficult and belongs to the so-called class of NP-complete problems. Moreover, according to the existing classification, the traveling salesman problem is an example of an optimization problem that is an example of the most complex subclass of this class.

In this paper, we describe several variants of algorithms for generating source data for the traveling salesman problem. We consider both the classical heuristics associated with the branch and bound method, and some added to them. Next, we present a software implementation of our interpretation of the algorithm. At the end of the paper, we formulate some tasks for further research, so the paper can be a project for students' scientific work.

Keywords: *optimization problems, traveling salesman problem, heuristic algorithms, branch and bound method, real-time algorithms, C++.*

Citation: B. F. Melnikov and E. A. Melnikova, "On the Classical Version of the Branch and Bound Method," *Computer tools in education*, no. 2, pp. 41–58, 2022 (in Russian); doi: 10.32603/2071-2340-2022-2-41-58

1. INTRODUCTION

The paper discusses the application of the branch and boundary method (BBM) to the solution of the traveling salesman problem (TSP). At the same time, in addition to the classic version,

some new heuristics are also considered, related to the need to develop real-time algorithms. More precisely, we consider such real-time algorithms that have the best (at the moment) solution at each specific moment of operation, and the user can view these pseudo-optimal solutions in real time, and the sequence of such solutions in the limit gives the optimal solution.

In addition, considering the example of the implementation of the algorithm of BBM for TSP we shall consider the *implementation* of discrete mathematics algorithms using object-oriented programming, and possibly the *general approach* to such an implementation

In this abridged version of the paper, we do not provide such possible sections:

— about different versions of generating input data for TSP;
— the description of our implementation of the classic version of BBM;
— about implementing auxiliary classes.

However, of course, the additions to the classic version of BBM and the implementation of the main classes are described in full.

For the implementation, we use the C++ language; this is the language that, without a doubt, "will take a clean first place" by any natural "complex criterion", including the following:

— first of all, of course, the ability to use object-oriented programming;
— the ease of design of complex data structures and the ability to use data abstraction;
— ease of implementation of algorithms, which are intended for working with discrete mathematical objects, for optimization problems, etc.;
— beautiful, clear, and transparent recording of the source code of programs;
— the efficiency of the resulting code of executable programs;
— the possibility of a small change in the program text, when changing (and sometimes significantly) the implemented algorithm;
— availability of translators and related development environments;
— convenience ("friendliness") of translators.

Of course, there are other single criterion "rating leaders", but we mean a "complex criterion".

When discussing the implementation of complex algorithms in discrete mathematics, it is important to note the following. Even very good books on algorithmization, including [4], have one common flaw; it can be briefly described as an insufficient description of the relevant programs "for real conditions", first of all for large dimensions. For example (but not only), we are referring to *memory allocation and deallocation issues* when working with very large dimensions. If the translator is responsible for memory allocation, then the executed code becomes significantly less efficient. This is confirmed comparison of the running time of complex programs written in different programming languages, in particular, programs for BBM for TSP considered in this paper. Therefore, we believe that the programmer should take care of allocating and freeing memory. And this is most convenient, apparently, also in C++.

There are many problems described in the literature that can be called *discrete optimization problems*; they are used in various fields of practical activity. Often these tasks are very difficult to solve on a computer, hence the name "intractable". More precisely, the possible approaches to the effective solution of these problems are difficult to solve (description of algorithms, programming); but the bulky solution (brute force method), as a rule, is programmed simply. On the other hand, almost every such intractable problem can be called a *mathematical model*; at the same time, both the model itself and the algorithms designed to solve it are often created for one subject area, but they can also be used in many other areas. An example of such a task (such a model!) is TSP ([1–3] etc.). The peculiarity of the problem is that with the relative simplicity of its formulation, finding the optimal solution it is very complex and belongs to the so-called class of NP-complete problems. Moreover, according to the classification given in [3] etc., the traveling

salesman problem is an example of an optimization problem that is part of the *most complex subclass*, i.e., the NPO(V) subclass.

Our goal is to discuss the implementation of the branch and bound method; the implementation of this method has a lot in common for a wide variety of discrete optimization problems, and we consider it on the example of TSP.

Here is the formulation of this problem. Given a complete graph (we shall usually consider directed graphs, "digraphs") of N vertices. Each its edge is marked with a non-negative number, we will always consider only integers. The task is to find a *loop* that passes through all N vertices of the graph, and this loop should have the lowest possible cost (the sum of the marks of its edges). An example of the input data (note that it is taken from [2]) is shown on Fig. 1, and the corresponding matrix representation is shown on Fig. 2.

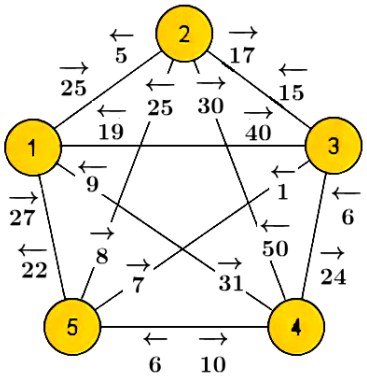


Figure 1. Source data

	1	2	3	4	5
1	999	25	40	31	27
2	5	999	17	30	25
3	19	15	999	6	1
4	9	50	24	999	6
5	22	8	7	10	999

Figure 2. Source data, the matrix

Let us go back to the algorithms. The traveling salesman problem, like most discrete optimization problems, can still be solved by the brute force method, but usually without much success. That is why the complicated algorithms are required for TSP, and the branch and bound method is one of them. It consists of several related *heuristics* — and in the monograph literature, such a "multiheuristic" BBM was apparently not previously noted. At the same time, the method itself can be applied-with very few changes, i.e., to many other discrete optimization problems.

2. MODERN ADDITIONS TO THE CLASSICAL APPROACH: ANYTIME ALGORITHMS AND THE RIGHT TASKS SEQUENCE

At the beginning of this section, we shall give a few words about so called *anytime algorithms*. They are real-time algorithms that have the best (at the moment) solution at each specific moment of operation, the user can view these pseudo-optimal solutions (also in real time), and the sequence of such solutions gives usually in the limit the optimal solution.

Apparently, more detailed definitions are not needed. But here is a possible *example* of the practical application for such anytime algorithms; this example, by the way, is suitable for any difficult problem. Thus, let us estimate the time to complete the entire task in 3 months, and the customer (the boss, the user) wants to get *at least some* acceptable solution much sooner; of course, it is desirable that it is more or less close to the optimal one. To say, he wants to obtain the first solution after 1 hour of the program. If the total time to complete the entire task is significantly more than 1 hour, then some solutions that are close to optimal begin to be obtained "almost" in 3 months, to say, in 2 months and 28 days. What should we do? That is why we should use some additional heuristics, which give even a "more distant" solution. (comparing the usual BBM), but very quickly. One of these heuristics (in fact, a modifications of BBM) is the use of so-called 1-trees. The other is so-called *right tasks sequence (RTS)*, see Fig. 3.

This heuristic is as follows. Each time we select the next separating element in some subtask (let it be T; we can also say "when getting the next right problem") we actually construct such a sequence when applying the heuristic:

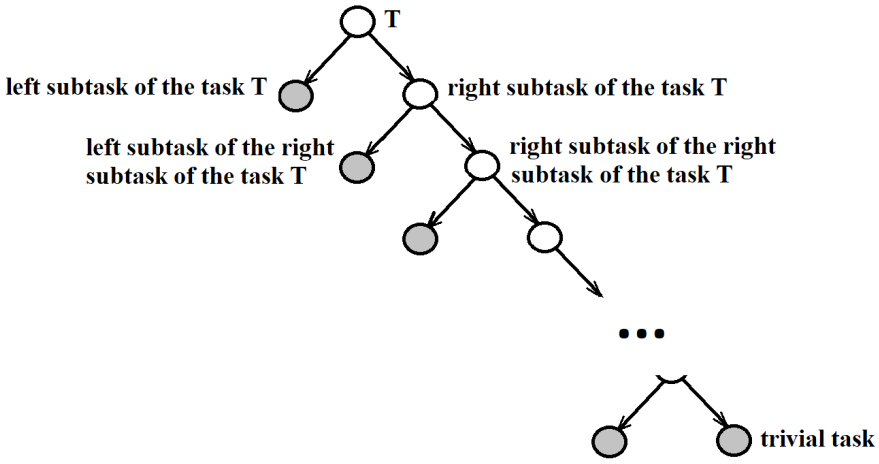


Figure 3. The right tasks sequence (the grey color shows the subtasks that are included in the list of subtasks for further solution)

— task T itself,
— the right (sub)task of the task T,
— the right task of the right task of the task T,
— etc.

Certainly, each time the corresponding left problems are constructed (and included in the list of problems for potential solution in the future):

— the left task of the task T,
— the left task of the right task of the task T,
— etc.

The described process ends:

— either when obtaining a trivial problem (for example, a zero-dimensional problem): in this case, we remember its solution (the boundary, the tour received at the time of its setting, and other *characteristics*) as *a pseudo-optimal solution of the current time* of the anytime algorithm;
— or when we get a sufficiently large boundary in any problem, for example, greater than the pseudo-optimal solution available at a given time.

Note that in practice, the described process of building a RTS *does not take much extra time* and *does not lead to a large increase of the list of tasks* intended for potential solutions in the future.

Thus, we have described a simple process of constructing an anytime algorithm based on some specific variant of the BBM; it is in fact the *truncated* branch-and-bound method. Next, we shall consider the order of iterating through the subtasks, which differs from the one proposed in [2]; this order follows from the algorithm for constructing the RTS given in the previous section. But in order to have something to compare it with, we shall use the same example of the given matrix (Fig. 2). We note that this example is interesting in this way: the optimal solution in it is *not where it was initially expected* (i.e., not in the set of tours that is initially considered as “more promising”).

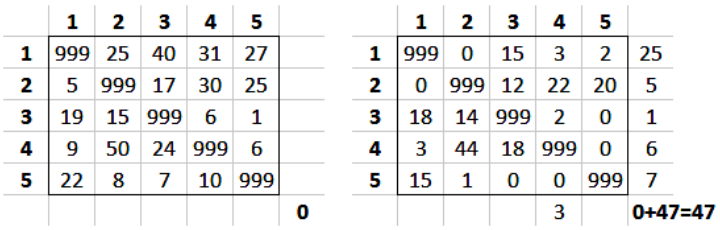
Below, we shall denote the arising tasks as follows:

— “base”: the given task will be denoted by ε (empty word);
— “step”: if some problem is denoted by X, then the left and right subtasks that arise from it are denoted by X_0 and X_1 respectively.

Thus…

Task ε: let us repeat source data in some modified format.

The next step, of course, coincides with the one discussed in [2]. We still have only one subtask, and we need to rank "zeros" for it, calculated their badness by [2]. We denote these values b_{ij}, the indices coincide with the indices of zeros:



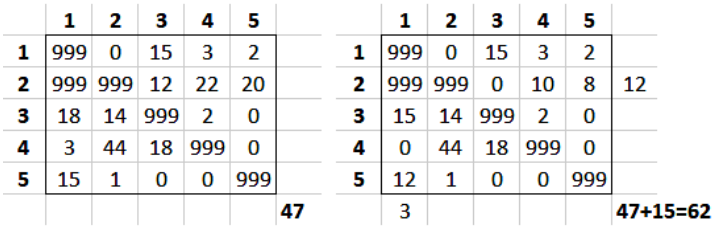
	1	2	3	4	5	
1	999	25	40	31	27	
2	5	999	17	30	25	
3	19	15	999	6	1	
4	9	50	24	999	6	
5	22	8	7	10	999	
						0

	1	2	3	4	5	
1	999	0	15	3	2	25
2	0	999	12	22	20	5
3	18	14	999	2	0	1
4	3	44	18	999	0	6
5	15	1	0	0	999	7
				3		0+47=47

Figure 4. Source data, task ε

$$b_{12}=2+1=3, \qquad b_{21}=12+3=15, \qquad b_{35}=2+0=2,$$
$$b_{45}=3+0=3, \qquad b_{53}=0+12=12, \qquad b_{54}=0+2=2.$$

Thus, the best zero corresponds to the element of the matrix with coordinates $(2,1)$, and based on this element, we form the first branching of BBM; after it, we get the following tasks 0 and 1. As we see, for the task 1, we can give one matrix only: no reduction is required for it, since each row and each column already has a zero. (However, we do not obtain "a large loop" for now; this is most easily explained by the fact that in the first row only one element is equal to 0, and this element, together with the previously selected one, would form "a small loop").



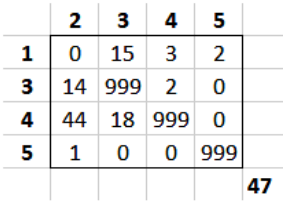
	1	2	3	4	5	
1	999	0	15	3	2	
2	999	999	12	22	20	
3	18	14	999	2	0	
4	3	44	18	999	0	
5	15	1	0	0	999	
						47

	1	2	3	4	5	
1	999	0	15	3	2	
2	999	999	0	10	8	12
3	15	14	999	2	0	
4	0	44	18	999	0	
5	12	1	0	0	999	
	3					47+15=62

Figure 5. Task 0

Starting from this point, the execution of the BBM goes "on a new path", which differs from [2]: we, regardless of anything, make the branching of the problem 1, it is the beginning of the construction of the RTS.

Let us continue this construction. To generate tasks 10 and 11 (as shown in Fig. 3, they are included in RTS), we must rank the zeros of the 1 problem. We shall use the badness notation b_{ij} for all matrices (similar to the notation used above); note that the indexes i and j are the row and column numbers of the *given* matrix shown in the figures to the left and top of the current matrices. Thus,

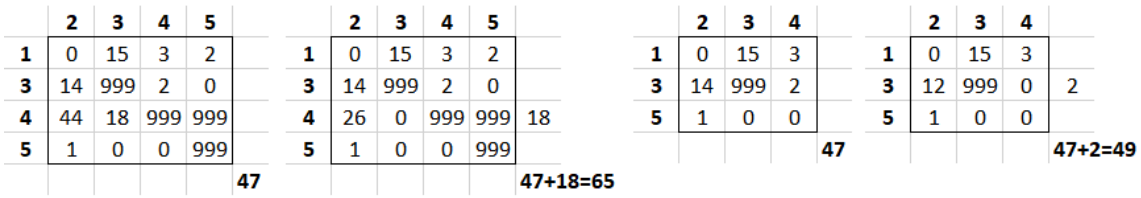


	2	3	4	5	
1	0	15	3	2	
3	14	999	2	0	
4	44	18	999	0	
5	1	0	0	999	
					47

Figure 6. Task 1

$$b_{12}=2+1=3, \qquad b_{35}=2+0=2,$$
$$b_{45}=18+0=18, \qquad b_{53}=0+15=15, \qquad b_{54}=0+2=2,$$

and we select an element with indexes $(4,5)$ for branching. Both the resulting 10 and 11 problems are shown in the following Fig. 7:



	2	3	4	5	
1	0	15	3	2	
3	14	999	2	0	
4	44	18	999	999	
5	1	0	0	999	
					47

	2	3	4	5	
1	0	15	3	2	
3	14	999	2	0	
4	26	0	999	999	18
5	1	0	0	999	
					47+18=65

	2	3	4	
1	0	15	3	
3	14	999	2	
5	1	0	0	
				47

	2	3	4	
1	0	15	3	
3	12	999	0	2
5	1	0	0	
				47+2=49

Figure 7. Task 10 (two matrices on the left) and task 11 (two matrices on the right)

Let us continue building RTS. Now, we need to rank the zeros in the 11 problem, and we get the following:

$$b_{12} = 3+1 = 4\,, \qquad b_{34} = 12+0 = 12\,,$$
$$b_{53} = 0+15 = 15\,, \qquad b_{54} = 0+0 = 0\,.$$

For branching, we select coordinates (5,3); the problems 110 and 111 resulting from this branching are given on Fig. 8. In the 111 problem, further branches are meaningless. Both zeros of the matrix of this problem form "the small loops", therefore, the current pseudo-optimal solution is formed by its non-zero elements, which, together with the zeros already selected earlier in the process of constructing RTS, really form the current pseudo-optimal solution. Its cost is 64 (see figure), and the loop is as follows:

	2	3	4	
1	0	15	3	
3	12	999	0	
5	1	999	0	
				49

	2	3	4	
1	0	15	3	
3	12	999	0	
5	1	999	0	
		15		49+15=64

	2	4	
1	0	3	
3	12	0	
			49

Figure 8. Task 110 (two matrices on the left) and task 111

$$1 \to 4 \to 5 \to 3 \to 2 \to 1\,.$$

The remaining unprocessed subtasks are 0, 10, and 110. Among them, we choose the problem 0 as having the smallest boundary 62 (in problems 10 and 110 the ones are 65 and 64). The next calculations are "not interesting": this bound *corresponds to the answer*, since there exists a sequence of zeros forming "a big loop":

$$1 \to 2 \to 3 \to 5 \to 4 \to 1\,.$$

The cost of this loop is less than all available limits, which makes it possible *to stop further calculations*.

3. GENERAL REMARKS ON THE PROGRAM, ITS DATA STRUCTURES AND ALGORITHMS

As we noted in the introduction, we try to use object-oriented features that are convenient for describing algorithms for solving mathematical problems, in particular, discrete optimization problems.

Let us go to the description of the organization of the whole program. In contrast to the fact that for the illustration of BBM is usually depicted in drawings, we do not use tree structures to organize the program, but we still have the tree in mind. In reality, instead of a tree, we store a list of its leaves only, and we "violate the ordering rules", which for the elements of this list can be derived according to [2]; see more detailed Section 5. However, any *natural* ordering algorithm can be considered as a special *heuristic*, and, apparently, only a practical study of program execution time can give an answer to the question which of these heuristics is better.

Formally, the leaves of the above-mentioned tree are most convenient to store as an array of pointers to subtasks; but, of course, other structures are possible (a list of such pointers, etc.). Of course, we need to consider an array with a dynamically changing dimension. There will appear "very many" subtasks (to say, up to 500 000 with the dimension of the problem about 100), so it is natural to place the data in dynamic memory.

Now the general work of our program becomes clear. First ("the basis" of the algorithm), based on the input data (which is just a matrix), a *task* is formed, containing *the array of subtasks*; initially, this array consists of one subtask only (it describes this input). However, each subtask is not only the matrix itself, but also some following additional information:

— the dimension of the (sub)task;
— the bound (0 for the initial subtask);
— row and column numbers of the subtask matrix (for the initialized subtask, they are from 1 to the dimension);
— the full information about already built part of the path (see also below, Section 4).

Secondly ("the step" of the algorithm):

1) we choose a task from the array (usually the first one);
2) if we do not need to finish it right away (i.e., usually in the case when its dimension is "not very small"), then:
   a) we select the separating element in it;
   b) we are trying to divide the selected task for this element on the left and right subtasks;
   c) if we managed to make such a division, then we form these subtasks, after that we include them "back" in the list;
3) otherwise, if we need to finish it right away, then:
   d) we finish it; we also get the current solution, as well as the corresponding tour;
   e) we compare the obtained solution with the current pseudo-optimal one;
   f) possible, we replace the current pseudo-optimal solution and the corresponding tour with new ones.

It is all! (This is the end of the informal description of the algorithm; and the end of its execution is an empty array of subtasks, i.e., the inability to perform sub-step 1.)

Next, we go to the direct description of the program, we give most of its text. First, we give the general constants for all functions and classes:

```
const int DIM_ALL = 99;          // the total dimension of the whole problem
const int DIM_ARR_SUB = 200000; // dimension of array of pointers to subtasks
const int INFTY = 999999;        // infinity for assignment
const int PEREBOR = 2;           // if the dimension is less or equal,
                                 // then we use brute force method
const int TURBO = 6;             // if the dimension is less or equal, then we
                                 // include in the beginning of array of subtasks
const int MAX_VALUE = 999;       // the maximum possible value of the matrix
```

Some of these constants are obvious (and the comments given in the program text are sufficient), but some other are still not completely clear.

DIM_ARR_SUB is the dimension of the array of subtasks (more precisely, pointers to subtasks). The array is stored in RAM, we do not save it to disk for simplicity. To get the results of computational experiments, we used the value 200 000, see Section 6.

PEREBOR; the meaning of this constant seems to be almost completely clear from a small comment in the program text. If the dimension is less than or equal to this constant, then a complete search (brute force method) is performed to solve the subtask; otherwise, it is the usual execution of the next step of BBM. In our program, the simplest option is installed, PEREBOR = 2, but readers should also try larger values if they want to (however, apparently, no more than 6). This should be offset by faster execution of the entire program.

TURBO; if the dimension is less than or equal to this constant, then we include the task *immediately after its forming* in the array, not in order of the border values, but *at the beginning*. This, among other things, makes it possible to implement as a very simple version of the construction of RTS, so are some other heuristics.

4. CLASS FOR SUBTASKS

Auxiliary classes are omitted in the description of the program; if necessary, we shall comment their use. This section describes in detail the class for the subtask, i.e., the most important and most complicated class of this project.

```
class SubTask {
private:
    int nDim;       // dimension of this subtask
    int nGran;      // boundary of this subtask
    Numbers* Lin;   // line numbers
    Numbers* Col;   // column numbers
    int* Matr;      // the elements themselves!
    Path* Next;     // the next city in the "already built" path
    Path* Prev;     // the previous city in the "already built" path
```

In the first picture, only its fields are shown. We do not need to comment on most of them (everything is clear on the basis of the above, the names of the fields themselves, as well as small comments in the program text); we shall give such ones only:

— the `Next` array is used to get a permutation of cities (often for pseudo-optimal solutions): if we go from the city `I` to the city `J` (i.e. we have established the need for such a trip), then we assume `Next[I]=J`;

— the `Prev` array is used for receiving the reverse permutation (to the permutation `Next`): if we go from the city `I` to the city `J`, then we assume `Prev[J]=I`.

Note once again, that the tree is “missing” in all classes! Both in this program and in similar ones, we storey its current leaf vertices onl. All this will need to be implemented in the main class `Task`. We shall look at this class further, but to understand the meaning of the `SubTask` class and its objects, it is desirable to know this information now.

Let us continue with the description of the `SubTask` class. Our implementation contains the following methods (see the figure). Comments to them are as follows.

```
private:
    int MakeIndex(int nX, int nY) { return (nX-1)*nDim + (nY-1); }
    void InitMemoryMatrix() { Matr = new int[nDim*nDim]; }
public:
    SubTask(int nDim);  // only memory allocation without value initialization
    SubTask(int* Matr); // initialization based on existing elements (nDimAll)
    ~SubTask();
    int GetLin(int I) { return Lin->Get(I); }
    int GetCol(int J) { return Col->Get(J); }
    void SetGran(int N) { nGran = N; }
    int GetGran() { return nGran; }
    int GetDim() { return nDim; }
    void Set(int nI, int nJ, int N) { Matr[MakeIndex(nI,nJ)] = N; }
    int Get(int nI, int nJ) { return Matr[MakeIndex(nI,nJ)]; }
    friend ostream& operator<<(ostream& os, SubTask& st);
private:
    int MinLin(int nX);     // minimum in string
    int MinLinDva(int nX);  // second by minimality in string
    int MinCol(int nY);     // minimum in column
    int MinColDva(int nY);  // second by minimality in column
    bool SetInftyByNumbers(int III, int JJJ);   // setting "inverse infinity"
```

- The second constructor `SubTask(int* Matr)` works only once (when initializing the entire task); and when generating a new subtask, we do not call it (see below for details). Of course, we usually initialize this first subtask using data taken from some text file; howe-

ver, there is only this data, i.e., elements of the matrix, and not all the elements of the subtask (and therefore we do not use stream input).
- `MinLinDva()` searches for the second smallest element in the string (this is necessary for the "optimal zero selection" option, see [2]); similarly for the column.
- `SetInftyByNumbers()`; with other implementations, this method may not be necessary; it sets infinity according to the row and column numbers *of the original matrix* (available in the number arrays), but not by the row and column numbers of the subtask matrix.

We omit the implementation of several simple methods. The rest will be considered in the same order as in the above description.

```
SubTask::SubTask(int nDim) {
    this->nDim = nDim;
    InitMemoryMatrix();
    Lin = new Numbers(nDim);
    Col = new Numbers(nDim);
    Next = new Path;
    Prev = new Path;
}
```

To the first constructor, apparently, additional comments are not needed.
Now the second constructor:

```
SubTask::SubTask(int* Matr) {
    this->nDim = nDimAll;
    InitMemoryMatrix();
    nGran = 0; // we have not calculated it yet
    Lin = new Numbers(nDimAll);
    Col = new Numbers(nDimAll);
    Next = new Path; Next->InitNull();
    Prev = new Path; Prev->InitNull();
    for (int i=1; i<=nDimAll; i++) {
        Lin->Set(i,i);
        Col->Set(i,i);
        for (int j=1; j<=nDim; j++) Set(i,j,Matr[MakeIndex(i,j)]);
            // from the parameter Matr to the field Matr
    }
    Reduction();
}
```

About the parameters of the constructors. Since the first constructor will be called constantly (each time a new subtask is generated-except for the very first one), the dimension to be set is unknown in advance, it is passed as a parameter of this constructor. on the contrary, the dimension is the maximum possible; as we have already noted, it is given by the dimension constant of the starting problem. For the `Matr` parameter, we use the `MakeIndex()` function, which was not originally intended for this type.

Let us continue to describe methods of `SubTask` class. `SetInftyByNumbers()` is setting infinity to prevent the formation of a "small loop":

```
bool SubTask::SetInftyByNumbers(int III, int JJJ) { // in the reverse order
    int nX = Lin->GetByNumber(JJJ);
    if (nX<=0) return false;
    int nY = Col->GetByNumber(III);
    if (nY<=0) return false;
    Set(nX,nY,nInfty);
    return true;
}
```

It is important to note the following:

— only one value ∞ is set, without the use of additional algorithms: they will already belong to the classes described below;
— the parameters are passed “in the right order” (row, then column), but used “in reverse one”: that is right, because we use setting the value ∞ for an element that is symmetric with respect to the main diagonal of the original matrix to the element under consideration;
— parameters are numbers of *initial* matrix, so we use the special function `GetByNumber()`.

Let us continue to describe the methods (see below). The comments are as follows.

```
    void ReductionLin(int nX);  // reduction by row
    void ReductionCol(int nY);  // reduction by column
    void Reduction();           // first by rows, then by columns
    bool BestNull(int& nXdel, int& nYdel);
    SubTask* MakeRight(int nXdel, int nYdel);
        // deleting nXdel and nYdel; we also modify ourselves for the left task!
    int Solve(Path*& Otvet);
        // we call for small dimensions only (uising PEREBOR);
        // Otvet: the answer; a new object is created!
};
```

- `Reduction()` is reducing string elements by a constant, see [2].
- `BestNull()` is selecting “the best zero”, see also [2].
- `MakeRight()` is apparently *the most important method of the whole project*; by the specified (previously selected) zero, it *constructs the right* subtask (generating it by calling the constructor and returning a pointer to it); after that, it *modifies itself for the left* subtask.
- `Solve()` is final solving a small-dimensional problem (after that, the resulting tour value will be compared with the current pseudo-optimal one, but this will already be done in the `Task` class); note in advance that we shall not give the text of the method, some explanations were already given when describing the general constants of the program.

Next is again the implementation.

```
void SubTask::ReductionLin(int nX) {
    int Min = MinLin(nX);
    if (Min==0) return;
    for (int j=1; j<=nDim; j++) {
        if (Get(nX,j)>=INFTY/2) continue;
        Set(nX,j,Get(nX,j)-Min);
    }
    nGran += Min;
}
```

For the reduction, apparently, no comments are needed, but the text of one such function we have given.

```
bool SubTask::BestNull(int& nXdel, int& nYdel) {
    int Max = -1;
    for (int i=1; i<=nDim; i++) for (int j=1; j<=nDim; j++) {
        if (Get(i,j)>0) continue;
        int nnn = MinLinDva(i) + MinColDva(j);
        if (nnn<=Max) continue;
        Max = nnn;
        nXdel = i; nYdel = j;
    }
    return Max >= 0;
}
```

Thus, `BestNull()` is the algorithm for selecting “the best zero” and returning the found pair of coordinates using the parameters; it is implemented exactly according to [2]. It is very important that the coordinates returned “like in the matrix“ (i.e., their possible values are from 1 to its dimension `nDim`), instead of “as in the original matrix” (i.e., from 1 to the maximum possible dimension `DIM_ALL`).

Now we turn to the most important method. “By and large” we can say, that this method is *the most important for the whole approach to implementing BBM* (for any discrete optimization problem, not just for TSP).

The text of this method is “very long” (which is understandable: we need to perform many different actions that are not related to each other), so we are again “dividing it into parts” in the figures. Just note that we shall return the constructed subtask using the `Return` pointer, and therefore we immediately use the `new` operator for it. Below, is the beginning of the method text.

Here and further in comments, we indicate the creation of the right subtask by point 1; it consists of several sub-items, i.e., auxiliary algorithms, which in the comments are entitled 1a, 1b, …, 1f. The code below shows only the simplest auxiliary actions needed to create the right subtask, they do not need additional comments.

```
SubTask* SubTask::MakeRight(int nXdel, int nYdel) {
    // 1) first, "from scratch", constructing the right subtask
    SubTask* Return = new SubTask(nDim-1);
    Return->SetGran(nGran);
    // 1a) installation the numbers of strings and columns
    for (int i=1; i<=nDim; i++) {
        if (i==nXdel) continue;
        int iNew = i<nXdel ? i : i-1;
        Return->Lin->Set(iNew,Lin->Get(i));
    }
    for (int j=1; j<=nDim; j++) {
        if (j==nYdel) continue;
        int jNew = j<nYdel ? j : j-1;
        Return->Col->Set(jNew,Col->Get(j));
    }
    // 1b) installation the values
    for (int i=1; i<=nDim; i++) {
        if (i==nXdel) continue;
        int iNew = i<nXdel ? i : i-1;
        for (int j=1; j<=nDim; j++) {
            if (j==nYdel) continue;
            int jNew = j<nYdel ? j : j-1;
            Return->Set(iNew,jNew,Get(i,j));
        }
    }
```

Next are more complicated actions, see the figure below. On it, sub-item 1c does not require comments, and sub-item 1d is the fact that in the right subtask, a trip between the selected pair of cities becomes mandatory. But the cities were selected by the row/columns numbers of the current matrix, so we select these numbers from the corresponding arrays `Lin` and `Col`. Next, we ut in the arrays `Next` and `Prev` the following city for the number `III` and the previous city for the number `JJJ`.

Subitem 1e is a call to set the replacement to infinity of the “impossible” zero; that is, one at which the corresponding movement is impossible due to the formation of a “small loop”. And subitem 1f is a search for all such “impossible” zeros that are obtained as a possible closure of “small loops” (the corresponding auxiliary algorithm is missing in [2]); however, this auxiliary

```
    // 1c) installing old paths
    Return->Next->InitCopy(Next);
    Return->Prev->InitCopy(Prev);
    // 1d) setting a new movement in them
    int III = Lin->Get(nXdel), JJJ = Col->Get(nYdel);
    Return->Next->Set(III,JJJ);
    Return->Prev->Set(JJJ,III);
    // 1e) setting the "inverse infinities" (combined with the following point)
    // 1f) possible setting "further inverse infinities"
    if (nDim>=3) { // ignore this sub-item if dimension of subproblem is small
        int JJJJJ = JJJ;
        for (;;) {
            if (!Return->SetInftyByNumbers(III,JJJJJ)) break;
            JJJJJ = Return->Next->Get(JJJJJ);
            if (JJJJJ<=0) break;
        }
        int IIIII = III;
        for (;;) {
            if (!Return->SetInftyByNumbers(IIIII,JJJ)) break;
            IIIII = Return->Prev->Get(IIIII);
            if (IIIII<=0) break;
        }
    }
```

algorithm can be dispensed with, since the absence of “small cloops” is automatically checked before considering the solution as acceptable and checking it for pseudo-optimality, see more in the `Task` class.

```
    // 1g) reduction (here, it is the complete one)
    Return->Reduction();
    // 2) now, we make the left subtask from "ourselves"
    Set(nXdel,nYdel,nInfty);
    ReductionLin(nXdel);
    ReductionCol(nYdel);
    // 3) end of function description
    return Return;
}
```

Everything given on this fragment of the `MakeRight()` function should not cause additional questions. The actions here are as follows:

— first, sub-item 1g: reduction of the obtained right subtask;
— secondly, sub-item 2: the record of “ourselves” as the left subtask;
— thirdly, also sub-item 2: reduction of the resulting left subtask by the row and the column; in this case, there exist the only row and the only column, besides their numbers are already known.

Next (see figure below), a simple description of the solution of a small-dimensional problem is given. We repeat that we only consider *a very simplified version*, that is, we only solve the problem to the end if its dimension does not exceed 2. At the same time, no more than 2 variants of cycles are possible; we check them to select the best one.

```
int SubTask::Solve(Path*& Otvet) { // the simplyfied version, for nDim==2
    Otvet = NULL;
    if (nDim>nPerebor) return nInfty;
    int I1 = Lin->Get(1), I2 = Lin->Get(2), J1 = Col->Get(1), J2 = Col->Get(2);
```

```
    if (Next->Get(I1)!=0 || Next->Get(I2)!=0 ||
        Prev->Get(J1)!=0 || Prev->Get(J2)!=0) return nInfty;
    // 1st variant of the loop
    int Sum1 = Get(1,1) + Get(2,2);
    // 2nd variant of the loop
    int Sum2 = Get(1,2) + Get(2,1);
    if ((Sum1 > nInfty/2) && (Sum2 > nInfty/2)) return nInfty;
    Otvet = new Path;
    Otvet->InitCopy(Next);
    int Sum = this->GetGran();
    if (Sum1<=Sum2) { Otvet->Set(I1,J1); Otvet->Set(I2,J2); Sum += Sum1; }
    else            { Otvet->Set(I1,J2); Otvet->Set(I2,J1); Sum += Sum2; }
    return Sum;
}
```

However, the above-mentioned simplification *does not affect the basic actions* required to perform more complex versions of the function (i.e., with an arbitrary value of the constant PEREBOR), namely:

— the comparisons of all resulting cycles,
— choosing the best of them,
— after that, return:

the cost of the best loop, “by the usual way”,
this loop, by the parameter.

5. CLASS FOR THE MAIN TASKS

First, let us consider “the interface part”, see the next figure.

```
class Task {
private:
    int* Matr;  // elements of the given(!) matrix, dimension is DIM_ALL
    int nKol;   // the current number of subtasks
    SubTask* Zadachi[DIM_ARR_SUB];
                // array of pointers to subtasks, indexing is "usual" for C
    int nOpt;   // the cost of the current pseudo-optimal solution
    Path* pOpt;
    Rezults rez;
public:
    Task();     // with the memory initialization
    ~Task();
    void ReadFile(int Nomer); // the number of the file; we fill in *Matr only
    void InitRnd();
    void InitFirst();   // the whole task (containing 1 subtask) is initialized
                        // (but not the matrix only)
    friend ostream& operator<<(ostream& os, Task& t);
    SubTask* ExtractFirst();
    bool SolveSubTask(SubTask*& st);
    bool AddSubTask(SubTask* st, bool bSimple); // if bSimple, including by #0
        // returning false occurs if there is an overflow of array of subtasks!
        // (but not in the case when we did not include the next subtask)
    void DelTail(int gran); // delete subtasks starting from this bound and more
    void Run();
};
```

Apparently, the fields are clear. Let us start the description of the headers of some methods “from the end” (from top to bottom). Run( ) executes the whole solution. To do this, we first create

an array of a single subtask (which is either entered from a file or initialized randomly), written to the array of subtasks. After that, we make the loop: we select the first subtask, extracting it from the array (in it, the tasks *usually* are ordered in ascending order of the boundary), and try to solve it using the methods of the SubTask class. In the case of a new solution (which, by the way, can not be called pseudo-optimal), we compare the cost of the tour with the existing pseudo-optimal one; if successful, we change corresponding pseudo-optimal solution and tour.

```
Task::Task() {
    Matr = new int[DIM_ALL*DIM_ALL];
    nKol = 0;
    nOpt = INFTY;
    pOpt = new Path;
}

Task::~Task() {
    delete[] Matr;
    delete pOpt;
}
```

We gave the obvious constructor and destructor, and the task initialization is as follows:

```
void Task::InitFirst() {
    nKol = 1;
    Zadachi[0] = new SubTask(Matr);
    nOpt = INFTY;
    pOpt->InitNull();
}
```

Next, extracting the subtask from the array and returning it (everything is also simple):

```
SubTask* Task::ExtractFirst() {
    if (nKol<=0) return NULL;
    SubTask* Return = Zadachi[0];
    for (int i=1; i<=nKol; i++) Zadachi[i-1] = Zadachi[i];
    nKol--;
    return Return;
}
```

Next, everything related to the completion of the subtask:

```
bool Task::SolveSubTask(SubTask*& st) {
    Path* otv; // "*otv" must be deleted!
    int nSolve = st->Solve(otv);
    if (nSolve >= nInfty/2) {
        if (otv!=NULL) delete otv;
        return false;
    }
    if (nSolve < nOpt) {
        nOpt = nSolve;
        pOpt->InitCopy(otv);
        rez.NewOpt(); // we noted that the step gave new pseudo-optimal solution
        DelTail(nSolve);
    }
    delete otv;
    return true;
}
```

Some comments are as follows.

- Is it necessary to finish solving the subtask? We do not solve this here; here, we simply perform such a final solution.
- In the current version, the option of passing the `SubTask* st` parameter is possible, but we consider the option, which is also suitable for the more general case.
- If we get at least some solution (not necessarily the new current pseudo-optimal one), then we return `true`; at the same time, at the "calling" level, this fact will be a sign that we can forget about the calling subtask.
- As we see from the method text, in the case when we still get a new current pseudo-optimal solution, we replace the associated fields of our class.
- Calling `rez.NewOpt ()` marks in the `rez` object the number of the step, at which this psudo-optimal solution was obtained; then, after the end of the whole process of solving the problem, we shall know the number of the last of these steps. (In general, the use an object of this class will become clear based on Section 6.)

The following figure shows adding a subtask to an array. Such comments are required:

```
bool Task::AddSubTask(SubTask* st, bool bSimple) {
    if (SolveSubTask(st)) return true; // there can be no overflow of the array!
    int nNewGran = st->GetGran();
    if (nNewGran >= nOpt) { delete st; return true; } // similarly
    // checking, whether we can increase the number of subtasks:
    if (nKol>=nDimArrSub) return false;
    int IndNew = 0; // index for inserting a subtask
    if (!bSimple) for (int i=0; i<nKol; i++) { // can we change this index?
        if (nNewGran <= Zadachi[i]->GetGran()) break;
        IndNew = i+1;
    }
    for (int i=nKol; i>IndNew; i--) Zadachi[i] = Zadachi[i-1];
    nKol++;
    Zadachi[IndNew] = st;
    return true;
}
```

- The problem to be solved is passed "on the pointer" by the first parameter.
- In the case when after the end of the method execution, we no longer need the subtask, we delete it using destructor. Otherwise, we include it back in the array of subtasks, i.e., the corresponding pointer "does not disappear". Thus, in any scenario, there will be no «holes» in the dynamic memory.
- The already considered `AddSubTask()` method returns `true` if and only if after its execution, the array of subtasks did not overflow (i.e., the method was completed successfully). In particular, in the case of solving a subtask, there can be no such overflow.
- Loop, not running if the corresponding response to the condition check `if (!bSimple)`, is perhaps *the most important heuristic*; as computational experiments show, *it significantly improves the overall solution time* compared to [2]. Sometimes (if, for example, the desired dimension has already been reached), the task must be added first!
- Almost the same statements are used for building RTS.

Method `DelTail()` is much simpler; it removes all the subtasks that have too large bounds. And the main method of the class (the "interface" one) is `Run()`. In it, `MakeRight()` is called, after which the resulting subtasks are processed. Note that the second parameter, which is always equal to the value `true` when calling the processing function for the resulting right subtask, shows the version of constructing RTS.

```
void Task::Run() {
    for (;;) {
        SubTask* ST1 = ExtractFirst();
        int I,J; if (!ST1->BestNull(I,J))
            { cout << endl << "UNEXPECTED END" << endl; break; }
        cout << "founding a pair: " << ST1->GetLin(I) << " " <<
            ST1->GetCol(J) << " (numbers of the given matrix)" << endl;
        SubTask* ST2 = ST1->MakeRight(I,J);
        if (!AddSubTask(ST1,(ST1->GetDim()<=nTurbo)))
            cout << "FAILED TO SAVE SUBTASK" << endl;
        if (!AddSubTask(ST2,true))
            cout << "FAILED TO SAVE SUBTASK" << endl;
        rez.NewIter(nKol);  // a new iteration has passed
                            // with (possibly) a new maximum number of subtasks
        cout << *this << endl;
        if (nKol>0) continue;
        cout << rez;
        cout << endl << "THE END" << endl; break;
    }
}
```

6. CONCLUSION. ON SOME RESULTS OF COMPUTATIONAL EXPERIMENTS

In this paper, we used a simple version of computational experiments. Here are the characteristics of the processor on which the computational experiments were carried out:

`Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.`

For the calculations, we used three dimensions (55, 80, and 99). The maximum possible value of the number of subtasks was always set to 200 000. The optimal solutions could be lost, but it is intuitively clear that increasing the number of subtasks by just 2 times for such dimensions and under our time constraints (see below) will make it possible to almost always get the best solutions. For each variant, we ran some computational experiments and recorded the following:

— the time of the complete solution in seconds; a limit of 3 hours was set which the program never exceeded; the column “time” below in the tables;
— the solution (it is unlikely to be of much interest); the column “solution” below in the tables;
— number of iterations (i.e., selections of the first subtask from the array); the column “iterations” below in the tables;
— maximum number of subtasks (which, according to the above, did not exceed 200 000); the column “subtasks” below in the tables;
— the number of the iteration at which the last pseudo-optimal solution was obtained; the column “optimum” below in the tables.

Dim=55	solution	iterations	subtasks	optimum	time
maximum	1766	56125	8058	8828	5,06
mediane	1580	12464	1634	2579	1,06
minimum	1494	7153	1034	52	0,742

Dim=80	solution	iterations	subtasks	optimum	time
maximum	1899	1595697	200000	871150	2267
mediane	1656	1240244	189998	116248	945
minimum	1523	108552	12395	77	18,7

Dim=99	solution	iterations	subtasks	optimum	time
maximum	1717	1874022	200000	1050808	8642
mediane	1585	1573722	200000	48316	5356
minimum	1467	321360	38475	96	95,2

All the values we needed were fixed in the `Rezults` class, which we mentioned above. We also recorded the maximum number of subtasks that were marked. For each of the characteristics listed above, we specify the maximum, median, and minimum values in the tables above.

To results of these tables, we add the following. For the dimension 55, we can guarantee that we have obtained the optimal solution in 100% of cases (since the value of the number of subtasks 200 000 has not been reached once). For the dimension 80, such "guaranteed" cases were about 60 %, and for the dimension of 99 were about 30 %. The obtained exact values of this value (the percentage of "guaranteed" cases) are hardly interesting: we have already noted that for "non-guaranteed" cases, the optimal solution is most likely always obtained, and with small additions to the program, this fact could be said with confidence.

References

1. M. Garey and D. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, San Francisco, CA, USA: W. H. Freeman & Co., 1979.
2. S. Goodman and S. Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, New York, NY, USA: McGraw–Hill, 1977.
3. J. Hromkovič, *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*, Berlin: Springer, 2004.
4. T. Cormen, Ch. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3th ed., Cambridge, MA: The MIT Press, 2009.

Received 23-05-2021, the final version — 22-07-2021.

Boris Melnikov, Doctor of Physical and Mathematical Sciences, Professor of the Faculty of Computational Mathematics and Cybernetics of Shenzhen MSU – BIT University, part-time Professor of the Faculty of Information Technologies of the Russian State Social University, ✉ **bormel@mail.ru**

Elena Melnikova, Candidate of Physical and Mathematical Sciences, Associated Professor of the Faculty of Information Technologies of the Russian State Social University, ya.e.melnikova@yandex.ru

Компьютерные инструменты в образовании, 2022
№ 2: 41–58
УДК: 37.01:007+004.82+004.85+519.713
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-2-41-58

О классическом варианте метода ветвей и границ

Мельников Б. Ф.[1], доктор физико-математических наук, профессор,
✉ bf-melnikov@yandex.ru, orcid.org/0000-0002-6765-6800
Мельникова Е. А.[2], кандидат физико-математических наук, доцент,
ya.e.melnikova@yandex.ru, orcid.org/0000-0003-1997-1846

[1]Совместный университет МГУ – ППИ, район Лунган, Даюньсиньчэн,
ул. Гоцзидасюеюань, д. 1, 517182, Провинция Гуандун, Шэнчжэнь, Китай
[2]Российский государственный социальный университет,
ул. Вильгельма Пика, д. 4, стр. 1, 129226, Москва, Россия

Аннотация

В компьютерной литературе описано много проблем, которые можно назвать задачами дискретной оптимизации: от шифрования информации в Интернете до поиска групп по интересам в социальных сетях. Такие задачи очень сложно решаются на компьютере и называются «труднорешаемыми». Точнее, сложно описывать возможные подходы к решению этих проблем; при этом программы, основанные на полном переборе вариантов, как правило, программируются просто, но работают значительно медленнее. Почти каждую из этих трудноразрешимых задач можно назвать математической моделью. Пример — задача коммивояжера. Особенность проблемы в том, что несмотря на относительную простоту её формулировки, поиск оптимального решения (оптимального маршрута) достаточно сложен. Эта задача очень трудна и относится к так называемому классу NP-полных задач, она является примером оптимизационной задачи из самого сложного подкласса этого класса.

В статье описывается несколько вариантов алгоритмов формирования исходных данных для задачи коммивояжера, рассматриваются как классические эвристики, связанные с методом ветвей и границ, так и некоторые дополнения к ним. Далее представлена программная реализация нашей интерпретации алгоритма, предложено несколько задач для дальнейшего исследования, поэтому статью можно считать описанием проекта для научной работы студентов.

Ключевые слова: *оптимизационные задачи, задача коммивояжера, эвристические алгоритмы, метод ветвей и границ, алгоритмы реального времени, C++.*

Цитирование: Мельников Б. Ф., Мельникова Е. А. О классическом варианте метода ветвей и границ // Компьютерные инструменты в образовании. 2022. № 2. С. 41–58. doi: 10.32603/2071-2340-2022-2-41-58

Поступила в редакцию 23.05.2021, окончательный вариант — 22.07.2021.

Мельников Борис Феликсович, доктор физико-математических наук, профессор факультета Вычислительной математики и кибернетики Университета МГУ – ППИ в Шэньчжэне, профессор факультета Информационных технологий РГСУ, ✉ bormel@mail.ru

Мельникова Елена Анатольевна, кандидат физико-математических наук, доцент факультета Информационных технологий РГСУ, ya.e.melnikova@yandex.ru