[3] Pollack, M., Solution of the k-best-routes through a network. — A review, *Journ. of Math Anal. Appl.*, 1961, 13:3, 547–559.

[4] Dreyfus S. E., An appraisal of some shortest path algorithms, *Operations Research*, 1969, 17:3, 345–412.

[5] Yen J. Y., Finding the K shortest loopless paths in a network, *Manag. Sci.*, 1971, 17, 711–715.

[6] Lawler, E. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem, *Manag. Sci.*, 1972, 18, 401–405.

[7] Shier, D. R. Iterative methods for determining the K shortest paths in a network, *Networks*, 1976, 6, 205–230.

[8] Romanovsky, J. V., Skurina, L. L., k-Optimal'nye reshenija v zadachakh diskretnogo i dinamicheskogo programmirovanija, In: *Issledovanie Operatsij i Statisticheskoe Modelirovanie*, St.-Petersburg, St.-Petersburg University, 1993. (in Russian, to appear)

[9] Minieka E., On computing sets of shortest paths in a graph, *Comm. of the ACM*, 1974, 17:6, 351–353.

[10] Minieka E., Shier D., A note on an algebra for the k best routes, *Journ. Inst. Math. Appl.*, 1973, 11:2, 145–149.

[11] Bertelo, U., Brioshi, F., *Non-Serial Dynamic Programming*, N.Y., Academic Press, 1972.

[12] Lawler, E., Comment on computing the K best paths in a graph, *Comm. ACM*, 1977, 20, 603–604.

[13] Romanovsky , J. V., Bui Dyk Thyk, Vychislitel'nye aspekty konechnogo dinamicheskogo programmirovanija, *Optimizatsija*, 1987, iss. 41(58), 148-162. (in Russian)

[14] Chen Y. L., Chin Y. H., The quickest path problem, *Computers and Operations Research*, 1990, 17:2, 153-161.

[15] Rosen, J. B., Su S.-Z., Xue G.-L., Algorithms for the quickest path problem and the enumeration of quickest paths, *Computers and Operations Research*, 1991, 18:6, 579–584.

[16] Dijkstra E. W., A note on two problems in connection with graphes, *Numerishe Math.*, 1959, 1, 269–271.

[17] Emelichev V. A., Diskretnaja optimizatsija. Posledovatelqnostnye shemy reshenija. *Kibernetika*. 1971, 6, pp. 109–121. (in Russian)

[18] Turner D. A., An overview of Miranda, In: *Research Topics in Functional Programming*, ed. by D. A. Turner, Addison-Wesley, 1990, pp. 1–16.

10.12 Example: Shortest paths in a loopless graph

Consider a graph $\langle M, N \rangle$ where $M = \{A, B, C, D, E, F, G\}$ and each arc in N is defined by the pair of the vertices and has the following lengths:

$$L_{AB} = 3, \quad L_{AC} = 1, \quad L_{BD} = 3, \quad L_{BE} = 12, \quad L_{CF} = 12 \quad L_{DE} = 6,$$

$$L_{DF} = 3, \quad L_{EG} = 2, \quad L_{FG} = 4, \quad L_{CD} = 4 \text{ and } 6.$$

This problem is written for the program as follows (the left column):

```
[A vG 0 ]                    vG   = [0]
[S vE vG 2                   vE   = (+2) vG
[S vF vG 4                   vF   = (+4) vG
[S aDE vE 6                  aDE  = (+6) vE
[S aDF vF 3                  aDF  = (+3) vF
[M vD aDE aDF                vD   = merge aDE aDF
[S aBE vE 12                 aBE  = (+12) vE
[S aBD vD 3                  aBD  = (+3)  vD
[M vB aBE aBD                vB   = merge aBE aBD
[S aCD1 vD 4                 aCD1 = (+4) vD
[S aCD2 vD 6                 aCD2 = (+6) vD
[M aCD aCD1 aCD2             aCD  = merge aCD1 aCD2
[S aCF vF 15                 aCF  = (+15) vF
[M vC aCD aCF                vC   = merge aCD aCF
[S aAB vB 3                  aAB  = (+3) vB
[S aAC vC 1                  aAC  = (+1) vC
[M vA aAB aAC                vA   = merge aAB aAC
[E
[T vA 4                      take 4 vA
```

The last line is the requirement to find the fourth path from A to G.

When I presented this paper to the seminar of Professor Vladimir Lifschitz at University of Texas (Austin), his student Norm McCain said to me about a functional programming language, Miranda, which is a language with *lazy evaluation* and has some of required constructions in ready. The right column shows the same example written in Miranda. Notice that according to [18], the use of Miranda should be considered more as a vehicle for the rapid prototyping of software, and it does not save us from developing of algorithmic support of enumeration processes, but the considence of ideas seems remarkable.

References

[1] Hoffman, W., Pavley, R., Method for the solution of the Nth best path problem, *Journ. of SIAM*, 1959, 6:4, 506–514.

[2] Bellman, R., Kalaba, R. On kth best policies, *Journ. of SIAM*, 1960, 8, 399–404.

10.11 The remainder

It remains to describe the main procedure and some variables that yet do not find their proper place.

```
procid mainproc;   /* it is the id of the main process */
cost   resval;     /* the resulting value of obj func  */
int    resdec;     /* the decision in the total call   */

main(argc,argv)
     int argc;
     char *argv[];
 {  int k;
    char t1[10],t2[10];

    printf("\n\nProcessing of k-optimal solutions\n");
    if (argc < 2)
      {
       printf("Usage: a.out <datafile>\n"); exit(4);
      }
    printf("data from file %s\n",argv[1]);
    filein = fopen(argv[1],"r");

    if (filein == NULL)
     {
       printf("Sorry, no such a file\n");
       exit(4);
     }

    SV_prep();
    printf("MAIN: end of preparation\n");

    gettoken(); printf("MAIN: Flag token %s\n",tok);
    gettoken(); strcpy(t1,tok); printf("Main: Call process %s\n",t1);
    gettoken(); sscanf(tok,"%d",&k);

    mainproc = findPRO(t1);
    SV_call(mainproc,k,&resdec,&resval);
    if (resdec == 0)
       printf("MAIN: solution number %d does not exists\n",k);
    else
       printf("MAIN: solution number %d has value %d with decision %d\n",
                 k,resval,resdec);

    exit(0);
 }
```

10.9 Addition process

The addition process has more complicated information field

```
struct ADP
   { procid LP; /* handle of left pr to be called   */
     procid RP; /* handle of right pr to be called  */
     int    LR[MAX_k]; /* Pareto boundary           */
   };
```

For the sake of simplicity, the Pareto boundary is defined here by an array that corresponds to each number of call of the left process its minimal mate in the right process. Thus we restrict the region of possible call numbers by `MAX_k`. This decision is simple and seems to be acceptable for the experimental needs.

The procedure of initialization of this process seems to be quite clear.

```
void ADD_prep()
{ struct ADP *adref;
  procid npr,lp,rp; int i;

  printf("ADD_prep\n");
  npr = regprocess(ADD);
  gettoken(tok); lp = findPRO(tok);
  gettoken(tok); rp = findPRO(tok);
  adref = (struct ADP *) malloc(sizeof(struct ADP));
  adref->LP = lp; adref->RP = rp;
  adref->LR[0] = MAX_k;
  for (i=1; i<MAX_k; i++) adref->LR[i] = 1;
  P[npr] = (void *) adref;
  return;
}
```

The form of input for the addition process is very similar to the merge process above:

```
 [+  add sh3 arr9
```

Here `add` is the name of the new process, that is followed by two names of the original processes.

10.10 Filter process

The implementation of filter processes hides inside it a serious difficulty, so I omitted this kind of processes and included a dummy procedure of initialization.

```
void FIL_prep()
{
  printf("FIL_prep: YET DUMMY\n");
  return;
}
```

The difficulty is that each specific filter process should be associated with its peculiar selection rule, although the rules can sometimes coincide. The rules should be given to the system separately, and it requires a special technique of software design that is hardly be discussed now.

```
[S  sh3 arr9 12
```

Here sh3 is the name of the process, arr9 is the name of the process called by it, and after that the shift constant is written.

10.8 Merge process

Each merge process has the following information field that is also allocated in the moment of initialization of the process

```
struct MGP
   { procid LP; /* handle of left pr to be called   */
     int    LN; /* its call number                  */
     procid RP; /* handle of right pr to be called  */
     int    RN; /* its call number                  */
    };
```

Here LP and RP are the handles of the processes to be merged, LN and RN are the number of their calls respectively made by this merge process.

The procedure of initialization of this process seems to be quite clear.

```
void MRG_prep()
{ struct MGP *mgref;
  procid npr,lp,rp;

  printf("MRG_prep\n");
  npr = regprocess(MERGE);
  gettoken(tok); lp = findPRO(tok);
  gettoken(tok); rp = findPRO(tok);
  mgref = (struct MGP *) malloc(sizeof(struct MGP));
  mgref->LP = lp; mgref->LN = -1;
  mgref->RP = rp; mgref->RN = -1;
  P[npr] = (void *) mgref;
  return;
}
```

The form of input for the merge process is as follows:

```
[M  mrg sh3 arr9
```

Here mrg is the name of the new process, that is followed by two names of the original processes.

It seems reasonable to presume a merging of an array of processes, or even of array of shifted processes. It makes no principal differencies, and it may be convinient in many problems of dynamic programming. Notice that such a process can be implemented with more sophisticated technique than a simple scanning or a stable search tree.

Here `arr1` is the name of the process, and after that five values are listed.

The procedure of initialization of this process consists mainly of a loop where the procedure `DB_save`- is called to save the current value pf the process in the data base.

```
void ARR_prep()
{ int r,npr,it=1;
  printf("ARR_prep\n");
  npr = regprocess(ARRAY);
  do
    { gettoken(tok);
      if (tok[0] != ']')
        { sscanf(tok,"%d",&r);
          printf(" num %d\n",r);
          DB_save(npr,it,it,r);
          it++;
        }
    } while (tok[0] != ']'); N[npr] = it;
  printf(" stop\n");
  return;
}
```

10.7 Shift process

Each shift process has the following information field that is allocated in the moment of initialization of the process

```
struct SHP
      { procid SP; /* handle of process to be called   */
        int    SN; /* call number                      */
        cost   AD; /* value to be added                */
      };
```

Here `SP` is the handle of the process to be shifted, `SN` is the number of calls made by this shift process, and `AD` is the shift constant.

The procedure of initialization of this process seems to be quite clear.

```
void SHF_prep()
{ int r; procid npr,rpr;
  struct SHP *shref;
  printf("SHF_prep\n");
  npr = regprocess(SHIFT);
  gettoken(tok); rpr = findPRO(tok);
  gettoken(tok); sscanf(tok,"%d",&r);
  shref = (struct SHP *) malloc(sizeof(struct SHP));
  shref->SP = rpr; shref->SN = 1;
  shref->AD = r; N[npr] = 0;
  P[npr] = (void *) shref;
  return;
}
```

The form of input for the shift process is as follows:

The procedure fills the gap between the last call of this process and the required call and after that calls the data bank to obtain the output values.

The main loop of this procedure consists of a switch that depends on the type of a performed process. This part of the process as well must be improved, and the actions with each type of processes must be hidden into special procedures.

10.5 Input procedures

The input is supported by two procedures. The first one gets the next line from the input flow, and the second procedure scans this string and writes the next token into string **tok** which is used for all calls of this procedure.

Thus we have

```
char   tok[10];    /* token for the input needs        */
char   a[200];     /* buffer to input a line           */
char  *at;         /* pointer inside  a                */

FILE *filein,*fopen();

int getline()
{
  int i;
  char *ac;

  for (ac = a,i=getc(filein); i != '\n'; ac++,i=getc(filein))
    *ac = i;
  *ac = '\0'; at = a;
  return 1;
}

int gettoken()
{
  char *tc;

  while (*at == ' ' || *at == '\0')
    if (*at == '\0') getline(); else at++;
  for (tc = tok; *at != ' ' && *at != '\0'; at++,tc++)  *tc = *at;
  *tc = '\0';
  return 1;
}
```

10.6 Array process

It becomes reasonable to combine a constant and an array in the same type. The array process was implemented in a direct way. We assume that in input stream this process is described as an ordered sequence that is finished by the token] as follows

```
[A  arr1 5 13 23 31 47 ]
```

```
            if (mgref->RN < 0)
              { t = -mgref->RN;   mgref->RN = t;
                SV_call(mgref->RP,t,&tdec,&tval);
              }
            else DB_find(mgref->RP,mgref->RN,&tdec,&tval);
            if (resdec != 0 &&
                (tdec ==0 || resval <= tval))
                { resdec = mgref->LN;
                  mgref->LN = -1 - mgref->LN;
              }
            else if (tdec !=0)
              {
                resdec = mgref->RN + MAX_k; resval = tval;
                mgref->RN = -1 - mgref->RN;
              }
            else resdec = 0;
            break;
          case ADD:
            adref = (struct ADP *) P[pi]; plr = adref->LR;

            for (t = 1,resdec = 0;
                 plr[t] > 1 || plr[t] < plr[t-1]; t++)
              if (plr[t] < plr[t-1])
                {
                  SV_call(adref->LP,t,&tdec,&tval);
                  if (tdec > 0)
                    {
                      SV_call(adref->RP,plr[t],&sdec,&sval);
                      if (sdec > 0 &&
                          (resdec == 0 || sval+tval < resval))
                        {
                          resval = sval+tval;
                          resdec = sdec*MAX_k + tdec; tbest = t;
                        }
                    }
                 }
            if (resdec != 0) plr[tbest]++;
            break;
          default:
            printf("SV_call: unexpected operation\n");
            resdec = 0;
        }
      if (resdec)
        {
          DB_save(pi,k,resdec,resval); N[pi] = k;
        }
    }
 if (k<=callno) *dec = 0;
 else DB_find(pi,callno,dec,val);
 return;
}
```

```
        if (tok[0] != '[')
           printf("SV_prep: unexpected token %s\n",tok);
        else switch(tok[1])
             { case 'E': return;
               case 'A': ARR_prep(); break;
               case 'S': SHF_prep(); break;
               case 'M': MRG_prep(); break;
               case '+': ADD_prep(); break;
               case 'F': FIL_prep(); break;
             }
      } while (1);
}
```

The procedure `SV_call` is definitely the main part of the system. Its input parameters are the handle of a required process and the number of required solution, and the output parameters are the value of the objective function and the decision associated with the solution. (Unfortunately, the part concerning the decision is not yet debugged).

```
void SV_call(pi,callno,dec,val)
     procid pi;      /* process handle */
     int    callno;
     int   *dec;
     cost  *val;
 { int k, t, tbest, lastcallno;
   struct SHP *shref;
   struct MGP *mgref;
   struct ADP *adref;
   int resdec,tdec,sdec; int *plr;
   cost resval,tval,sval;
   printf("SV_call proc %s/%d\n",C[pi],callno);
   lastcallno = N[pi];
   for (k=lastcallno+1,resdec=1; k<=callno && resdec != 0; k++)
     {
       switch(T[pi])
          {
           case ARRAY:
              resdec = 0; break;
           case SHIFT:
              shref = (struct SHP *) P[pi];
              SV_call(shref->SP,shref->SN,&resdec,&resval);
              if (resdec != 0)
                {
                  resval += shref->AD; shref->SN++;
                }
              break;
           case MERGE:
              mgref = (struct MGP *) P[pi];
              if (mgref->LN < 0)
                { t = -mgref->LN;   mgref->LN = t;
                  SV_call(mgref->LP,t,&resdec,&resval);
                }
              else DB_find(mgref->LP,mgref->LN,&resdec,&resval);
```

```
     int   PROtype;
{ int p; p = PRO_k++;
  gettoken(tok); T[p] = PROtype;
  C[p] = PRO_pos;
  strcpy(PRO_pos,tok); PRO_pos += strlen(tok)+1;
  printf("RP: Process %s is saved with number %d\n",tok,p);
  return p;
}

procid findPRO(name)
     char  *name;
{ int p;
  for (p=1; p< PRO_k; p++)
   if (strcmp(name,C[p]) == 0) return p;
  printf("fP: Process %s is unknown\n",name);
  return 0;
}
```

The procedure `regprocess(t)` includes the process of a given type into the system, it receives its name from the input flow (it is a temporary solution, the name should a parameter) and returns the handle of the new process. Later other procedures fill the enties of other arrays associated with the process, this part of the program should be also improved.

`findPRO` finds the handle of a process with a given name.

Since the information about processes is yet saved in arrays of free access, the operations with the data are not yet collected into an abstract type of data, but surely it should be done.

10.4 Supervisor

The supervisor consists of two procedures, rather independent. The first procedure serves to prepare the initial data, i.e., to initializate the data bank and the bank of processes, and to fill it by initial information. Here we selected the input from a file that is readed by separate lines, and each lines in diviled into tokens including no spaces. The description of each process started with a token with `[` as its first character, and the next character serves to switch. Depending on this letters one of procedures that initialized a process is called. The initialization is finished when a token `[E` appeares.

```
void SV_prep()
{
  printf("SV_prep: Debugging\n");
  T = (int *) malloc(PRO_lim*sizeof(int));
  N = (int *) malloc(PRO_lim*sizeof(int));
  P = (void **) malloc(PRO_lim*sizeof(void *));
  C = (char **) malloc(PRO_lim*sizeof(char *));
  PRO_pool = (char *) malloc(PRO_poollim*sizeof(char));
  PRO_pos = PRO_pool; PRO_k = 1;
  DB_init();
  getline();
  do { gettoken(tok);
```

10.2 Data bank of calculated values

An important part of our system is the data bank that collects all calculated values of all processes involved. Here we restricted ourselves by an elementary implementation of the bank as an array of records, so only the structure of this record and the headers of procedures are of interest

```
struct DB_el
  { procid p;  /* handle of the process           */
    int    n;  /* call number                     */
    int    d;  /* optimal decision                */
    cost   v;  /* value of O.F.                   */
  };

void DB_init();

int DB_save(pi,callno,dec,val)
   procid pi;
   int    callno;
   int    dec;
   cost   val;

void DB_find(pi,callno,dec,val)
   procid pi;
   int    callno;
   int   *dec;
   cost  *val;
```

The procedure `DB_init` opens the data bank, `DB_save` saves a new record and returns 1 in the case of failure, `DB_find` finds the record with given key arguments (handle of a process and the number of its call) and fills a given place by the information from this record.

10.3 Bank of processes

The processes involved in the system are also to be saved in a special bank. Here we used a system of several arrays, and in the most of them the handle of the process is simply the index of the process in these arrays. The only exclusion is the pool of the names of processes, where we assumed a special array of pointers to the names.

```
int   *T;          /* type of the process             */
int   *N;          /* number of saved calles          */
char  **C;         /* ref to the process name         */
void  **P;         /* ref to the process info         */
int    PRO_k;      /* no of processes                 */
char  *PRO_pos;    /* pos of free space in PRO_pool   */
char  *PRO_pool;   /* pool of names of processes      */

procid regprocess(PROtype)
```

Each of P'_k is in fact the set of permutations of indices k, $k+2:n$, and the value of the objective function consists of the constant term $\alpha_k = \sum_{i<k} a_i b_i + a_k b_{k+1}$ and a variable term $\sum_{i>k} a_i b_{\pi(i)}$.

Thus the second after optimum solution is obtained by merging processes associated with the sets P'_k. In their turn, each process P'_k is a process P_S actually of the same type as the whole process but with another set of indices S and shifted to α_k. The process P_S is new born.

On the subsequent steps each selected permutation $\overline{\pi}$ generates a similar set of processes, although the processes generated earlier can be reused. These processes are defined as

$$P_{\overline{\pi},k} = \{\pi \mid \pi(i) = \overline{\pi}(i),\ i < k,\ \pi(k) > \overline{\pi}(k)\}$$

with

$$P'_{\overline{\pi},k} = \{\pi \mid \pi(i) = \overline{\pi}(i),\ i < k,\ \pi(k) \text{ is the next after } \overline{\pi}(k)\}.$$

A similar splitting of processes is applicable to other problems where the feasible solutions are permutations, for instance to the assignment problem.

10 The experimental software

The goal of this section is to consider in a simplest way all necessary construction that allow us to write simple algorithms, and to execute them in an interpretation mode. Some weak points of this program were clear from the beginning. However, such a preliminary scheme can be also interesting as the background for more professional next version.

10.1 Definitions and data structures

Let us start with some general definitions and numerical parameters

```
#define cost int
#define procid int
#define ARRAY  1
#define SHIFT  2
#define MERGE  3
#define ADD    4
#define PRO_lim 100
#define PRO_poollim 1000
#define MAX_k  100
```

We defined here special names `cost` for the units of the objective function and `procid` for the handles of processes, both to save a possibility to replace the units and to see better the types of identifiers.

All other strings are the temporary definition of the types of processes (they look better in an enumeration type), and some numerical limits for the experimental software.

Thus, let a directed graph $\langle M, N\rangle$ be given, l_j be the length $j \in N$, and it be required to find several shortest paths from m_0 to m_+.

First, let us find the shortest path. Arrange an empty container for each vertex $i \in M$, and put the zero value into the container for m_0. Each container will be in one of two states, *active* or *passive*. Initially all containers are active.

The search of the shortest path consists of several iterations. On each iteration we select the container with the minimal current value among active nonempty containers. Let it correspond to a vertex i_t. If $i_t = m_+$, the calculation is finished. In the other case consider all arcs $j \in N$, $b(j) = i_t$, and for each j put $l_j + v(i_t)$ into container $e(j)$. Make container i_t passive.

This scheme differs from Dijkstra's in one point: his containers include only one value each, currently the best. The container in Shier's algorithm [7] contains a predescribed number K of the best values for each vertex, since it uses the technique of arrays.

When the first path is found, all other paths are produced by a standard method:

a) activate all containers along the previous path;

b) repeate the previous procedure with active containers until the container m_+ will be processed.

To enumerate the loopless paths in a graph with contours, it is enough to assume a filter that selects the loopless paths before their saving into a container. It is enough to check that the end vertex of the path does not contained in any other place of the path.

9 Splitting of processes

When a discrete optimization problem is solved by a method different both from the Branch-and-Bound technique and dynamic programming, it is necessary to search for another waya and another kinds of processes.

In some problems the optimal solution is obtained applying special procedures, for instance, greedy algorithms or methods of linear programming. The deletion of a single element from the set of feasible solutions changes the properties of the problem. However, as it was marked by V. A. Emelichev [17], this remaining set often may be splitted into a number of subsets obtaining the same property regarless the algorithm as the initial set.

As an example, consider the problem of finding of permutation π of components of a vector $b = (b_1, \ldots, b_n)$ that minimizes the sum $\sum_{i\in 1:n} a_i b_{\pi(i)}$ (or enumeration of permutation in order of increasing this function). The solution of the minimizing problem is as follows: If $a_1 \geq \cdots \geq a_n$ and $b_1 \leq \cdots \leq b_n$ select the permutation $\pi = (1, 2, \ldots, n)$.

Deleting this permutation, we can split the remaining set into the following sets

$$P_k = \{\pi \,|\, \pi(i) = i,\ i < k,\ \pi(k) > k\}.$$

It is easy to see that search of the next best permutation may be restricted by the sets

$$P'_k = \{\pi \,|\, \pi(i) = i,\ i < k,\ \pi(k) = k + 1\}.$$

The second modification is that it will be more convenient to find nodes and edges that should be deleted. Their number is $|N| - k = d$, both for edges and nodes, and our goal is to delete them by such a way that the weight of edges to be deleted is minimal.

The selection of the node i_1 as the root of the tree determines the orientation of the tree from the root to leaves. Now we can talk about beginning and final nodes of each edge.

Determine now the system of Bellman functions $V_i(t)$ for nodes with $0 \le t \le K_i$ and $v_j(t)$ for edges by the following recurrent relations:

$$V_i(t) = \max \left\{ v_j(\tau_j) \;\middle|\; b(j) = i, \sum\nolimits_j \tau_j = t \right\},$$

$$V_i(0) = 0; \qquad K_i = 0 \;\text{ for leaves},$$

$$v_j(t) = \begin{cases} V_{e(j)}(t), & t \le K_{e(j)}, \\ V_{e(j)}(t-1), & t = K_{e(j)} + 1. \end{cases}$$

The upper bounds K_i of domains of V_i are defined as 0 for leaves and as $\min\{d, \sum_{j,\, b(j)=i} K_{e(j)}\}$ for all other nodes.

Since all operations that present in these relations are of types described above, we are able to construct the enumeration process. Note that although it is too obvious, a direct its consideration, or enumeration algorithm seems to be rather tedious.

For the general scheme of Bertele-Brioshi elimination, the same approach is valid, but with yet more complicate implementation of the scheme.

7 Container processes

Now let us introduce a new type of process, a *container*. It receives by a special operation some values and in each call returns the lowest value (deleting it from the container). For the correctness of its work it is necessary to provide that each newcoming value is not less than the value returned by the process in the last call.

In a distinguish with other processes, we assume a possibility of a preliminary call of a container to get it current value without changing its internal state.

This process is required to describe the enumeration of the kth best paths in a graph that is based on the famous Dijkstra algorithm to find the shortest path.

8 Dijkstra method for the shortest paths

The Dijkstra algorithm [16] seems to be well known, so let us move directly to its modification.

It is well known that the algorithms are more simple in the case if the loops in the paths are admissible, and we start with it.

The original problem is to minimize the value of S on the direct product of A_j. In our consideration, we want to construct a process that enumerates elements of $\mathsf{X}_{i\in M}A_i$ by ascending the values of objective function S.

The Bertele-Brioshi approach consists of a sequential elimination of variables a_i. Namely, each variable a_i is represented as a function of neighbor variables. This function is obtained in result of solving (by enumeration) of a parametric extremal problem where the objective function is the subsum of S with functions that depend on a_i and all other involved variables are considered as the parameter. In the case of a tree, the order of eliminations can be chosen by such a way that the tree-like structure remains after each elimination.

We shall do the same, but with the opposite direction of simplification of the graph. Choose an arbitrary node $i_1 \in M$ as a root, and consider the star of edges incident to i_1. Let $R(i_1)$ be the set of nodes that belong to these edges, and $q = R(i_1)$. The deletion of $R(i_1)$ splits the graph on $q+1$ components, namely, the node i_1 itself and q trees, each containing precisely one node from $R(i_1)$. Let T_j be the tree containing the node $j \in R(i_1)$, and j be considered as its root.

Now we can represent the objective function as

$$S = \sum_{j\in R(i_1)} (f_{i_1 j} + S_j), \quad \text{where} \quad S_j = \sum_{e\in T_j} f_e.$$

In its turn, each sum S_j can be considered as an objective function for a similar problem for subtree T_j.

Minimization in the original problem may be considered as two-level minimization. Its external variable is a_{i_1}, and all other variables are for internal minimization

$$\min\{S \mid \{a_i\}_{i\in M}\} = \min_{a_{i_1}} \min\{S \mid \{a_i\}_{i\neq i_1}\} = \min_{a_{i_1}} \sum\nolimits_{j\in R(i_1)} \min\{f_{i_1 j} + S_j \mid \{a_i\}_{i\in T_j}\}.$$

Now it is clear that the required process can be arranged as a merging of several processes $\mathcal{P}_a$ corresponding to various values a of the variable a_{i_1}.

In its turn, each process $\mathcal{P}_a$ is obtained as a sum of processes corresponding to summands in the formula, because when the value a_i is fixed, the sets of variables in summands become mutually independent. For each summand we have a shift $f_{i_1 j}$ and a similar problem for the tree T_j.

Thus the case of nonserial dynamic programming requires only familiar operations.

Consider one application as an example.

Let a nondirected tree $\langle M, N\rangle$ be given, and each edge $j \in N$ have a positive weight w_j. Find a subtree $\langle M', N'\rangle$ with a prescribed number of nodes k and with maximal total weight of arcs $\sum_{j\in N'} w_j$.

This problem is easily solved by the method described above after certain modifications of the problem.

First, we can select a node i_1 and find the subtree with assumption that $i_1 \in M'$. We can always select such a node a priori if $k > |N|/2$. If k is not large enough then we can consider separately the subtrees that remain after deletion of i_1 and its star.

This set of definition is actually a program which can be executed to obtain the result (see subsection 10.12 below).

5.2 The quickest paths

This problem was considered by Y. L. Chen and Y. H. Chin [14] and later discussed by J. B. Rosen, S.-Z. Su, and G.-L. Xue [15]. The authors considered a graph $\langle M, N\rangle$ where each arc j is associated with two numbers, l_j and w_j, and the objective function is defined on the set of paths from m_0 to m_+ as follows

$$v(p) = \sum_{j\in p} l_j + \max\{w_j \mid j \in p\}.$$

Since we need here only a simple example, let us consider this problem for a loopless graph. The set P of all possible paths from m_0 to m_+ can be partitioned as

$$P = \bigcup\nolimits_{j\in N} P_j,$$

where P_j consists of all paths where j has the maximal w_j and is the first among arcs with the same value of w.

The process for the set P is constructed by merging the processes for nonempty P_j. The objective function for all paths of the same set P_j consists of three summands. One of them is the constant term $l_j + w_j$, the second corresponds to the length of a path from m_0 to $b(j)$ that uses only arcs with $w < w_j$, the third corresponds to the length of a path from $e(j)$ to m_+ that uses only arcs with $w \le w_j$. We can produce the processes enumerating both types of paths, find the sum of them and shift it on $l_j + w_j$. Actually we need to consider only nonempty sets of the partition mentioned above.[2]

6 Nonserial dynamic programming

U. Bertele and F. Brioshi [11] proposed a scheme of excluding variables in optimization problems that they called as *nonserial dynamic programming*. This scheme may be considered as a helpful extension of the original Bellman's scheme. Here we consider only one of their simplest schemes; however, this one is important for applications.

Let a nondirected tree $\langle M, N\rangle$ be given. Each its node is associated with a nonempty finite set A_i, and each edge (i, j) with a function $f_{ij}(a_i, a_j)$ that is defined on the set $A_i \times A_j$. Consider the sum

$$S(\{a_i\}_{i\in M}) = \sum_{(i,j)\in N} f_{ij}(a_i, a_j).$$

[2] It is not difficult to enumerate the arcs for whose the sets of paths are nonempty. The procedure is as follows: find a path from m_0 to m_+, find the arc with maximal w_j on this path. Delete it from the graph. Repeat this step while a path from m_0 to m_+ exists.

and two other elements $(k+1,l)$ and $(k,l+1)$ are considered. Each of them is to be included into $\partial C'$ if it is not preceded by the present elements of $\partial C'$.

This operation enables us to consider more general optimization tools, for instance, for branching schemes of dynamic programming, including mentioned above nonserial dynamic programming, as it was introduced by U. Bertele and F. Brioshi [11].

Some other operations with processes will be mentioned below.

4 Coordination of calls

The manipulations with processes requires a certain coordination of the calls, since one and the same process α may be called by several other processes.

If these processes act independently, and each of them acts in assumption that it calls the process α solely, it should enumerate indeed all results of α, while independent calls may dismiss it. A software tricks allow us to provide each calling process by its own copy of α, but it leads to a chain reaction of extra calls. To exclude this misunderstanding and/or repeated calls of the process, we introduce a *supervising system* that provides the correctness of the calls. In particular, this supervisor saves all information about previous calls of each process in a special data base.

If a current call asks for the information that was given before to any other process, the supervisor finds the information in the data base and returns it.

If the call requires a new information, the supervisor arranges all intermediate calls and saves all results. Notice that this supervisor may be a global construction that controls the calling of all processes simultaneously and arranges parallel execution of processes.

5 Some elementary examples

5.1 The shortest paths in a loopless graph

It is the simplest problem of dynamic programming. We consider here its solving by a *cascade method* [13] which is based on a direct use of the recurrent Bellman relation.

Let $\langle M, N\rangle$ be a contourless directed graph with the set of vertices M and the set of arcs N. Each arc $j \in N$ is associated with a positive number l_j that is the length of this arc. Let two vertices m_0 and m_+ be given. Find the process that enumerates all paths from m_0 to m_+ by ascending of their lengths.

Denote the process that enumerates all paths from $i \in M$ to m_+ by α_i. Let N_i^+ be the set of arcs that exit from i, and $e(j)$ be the end vertex of the arc j. If β_j be the process enumerating all paths to m_+ starting from the arc j, then we have obviously

$$\alpha_i = \text{MERGE}\{\beta_j \,|\, j \in N_i^+\}, \qquad i \neq m_+, \qquad \alpha_{m_+} = \text{CONST}\{0\},$$

$$\beta_j = \text{SHIFT}\{\alpha_{e(j)}, l_j\}.$$

2°. *Filtering.* Let a subset $A' \subset A$ be given. Then the process $\mathcal{P}(f, A')$ may be constructed from $\mathcal{P}(f, A)$ trough calling it until an element from the A' is achieved. This element is then returned as the result of the given call of $\mathcal{P}(f, A')$.

3°. *Merging.* Let two processes, say $\alpha = \mathcal{P}(f, A)$ and $\beta = \mathcal{P}(g, B)$ be given. For the sake of simplicity, assume that the sets A and B have nothing in common, i.e., we want exclude yet the case when $A \cap B \neq \emptyset$ and the functions f and g coincides on this intersection. Our goal is to define a process $\gamma = \mathcal{P}(h, A \cup B)$, where h coincides with f and g respectively in their domains.

The required process can be constructed as follows:

preliminary: Call α and β and save the returned values;

in each call: Compare the saved values v_α and v_β and get the minimum $v_{\min}$. Call once more the minimand process and update its saved value. Return $v_{\min}$ as the result.

Notice that this operation is well associated with the merging of ordered files, the only difference that it acts with processes of unspecified nature and produces also a process that can be considered as an input for other operations (including possibly a next merging). [1]

In the case if $A \cap B \neq \emptyset$, we can consider the sets A and $B \setminus A$. The filter operation allows us to transform the process β into $\mathcal{P}(f, B \setminus A)$.

The merging may be obviously generalized to the case of several processes, say, for $\mathcal{P}(f_1, A_1)$, ..., $\mathcal{P}(f_k, A_k)$. The required process selects every time the minimal value among the current returns of these processes. The organization of this search may be different, depending on a concrete situation, it may be an ordinary scanning, an unchangeable tree, a heap, or something else.

Actually, it is the merging that provides the choice of a minimum, and thus the process-based solving of extremal problems.

4°. *Addition of processes.* Let two processes, say, $\alpha = \mathcal{P}(f, A)$ and $\beta = \mathcal{P}(g, B)$ be given. Consider the set $C = A \times B$ and define on C a function $h(x, y) = f(x) + g(y)$. Our goal is the process $\mathcal{P}(h, C)$. We shall refer to the cardinality numbers of elements of processes α and β in their calling, let $\#_\alpha x$ denotes the number of call of the process α in which the element x appears, and similarly for process β.

Let C' denote the set of elements of C that were not yet called by the enumeration procedure. Introduce a partial order on C':

$$(x_1, y_1) \prec (x_2, y_2) \leftrightarrow \#_\alpha x_1 \leq \#_\alpha x_2 \ \text{ and } \ \#_\beta y_1 \leq \#_\beta y_2.$$

Let $\partial C'$ be the Pareto boundary of the set C' in the sense of this partial order, i.e., $\partial C'$ be the set of elements mutually incomparable by $\prec$ and such that each element $z \in C \setminus \partial C'$ is preceded by an element $z' \in \partial C'$. It is clear that the minimum element of C' at every call of the process $\mathcal{P}(h, C)$ must be taken from $\partial C'$.

We can easily save and update the information about $\partial C'$, since initially it consists of the sole element $(1, 1)$, and after each call the selected element (k, l) is excluded from $\partial C'$

[1] E. L. Lawler [12] mentioned the merging of two ordered arrays as an efficient tool in solving problems of k-best paths.

not match the actual conditions. For this reason, it is more desirable not to solve a given problem, but rather to have a possibility to vary its conditions.

As an approach to varying the model, we consider here a rather modified problem, that is to find several best elements of A in the sense of values of the objective function f. This modification arose in the literature many times, here we propose mainly a new approach which seems to be more adequate to the problem and allows us to describe the algorithms in simpler terms.

Namely, our goal is to construct a computational object, *a process*, that may be *called* and, answering these calls, it works out the elements of the set A by ascending the values of the objective function f, starting with the minimum.

However, we do not only construct a single process for the given problem, but consider the variety of all possible processes of that kind and study the operations that allow us to construct new processes from existing ones. This approach is well agreed with such usual methods of discrete optimization as sequential partitioning of a problem to subproblems and recursion. At the same time, some special methods of optimization (such as greedy algorithms when they provide the optimal solution, or Hungarian method for the assignment problem) lead us to some difficulties, although here we propose some first glance ideas.

3 Operations with the processes

The most of proposed operations are quite obvious. Nevertheless they give us the background to very powerful algorithms.

As the background for our construction we use *elementary processes* that simply enumerate the elements of finite sets (arrays), maybe after some their preparation. We use the elementary processes to solve the mentioned extremal problem when the set a consists of a modest amount of elements. As a more advanced version of an elementary process we can consider any algorithm that produces a sequence of nondecreasing numbers.

After exhausting of the set, the elementary process (and each one in the remainder) returns a special *infinite* value.

1°. *Superposition of processes.* Let a nondecreasing function g be defined on the set $f(A)$ of possible values of function f. Associate with this function the extremal problem:

Minimize $g(f(x))$ for $x \in A$.

Let $\mathcal{P}(f, A)$ denote the process that enumerates the elements of the set A by ascending the values of function f. Then a standard step (*a call*) of the process $\mathcal{P}(gf, A)$ can be obtained as follows:

call $\mathcal{P}(f, A)$ to receive a current element $x \in A$ and $\phi = f(x)$;
calculate $\gamma = g(\phi)$;
return x and γ.

Notice that the most trivial partial cases of linear function with a positive multiplier, and, in particular, a constant shift of the value are very important.

1 Introduction

The enumeration of solutions next after optimum is a natural generalization of an ordinary extremal problem in the case when the set of feasible solutions is discrete. The first algorithms of that kind were considered yet by W. Hoffman and R. Pavley [1] and by R. Bellman and R. Kalaba at the very beginning of dynamic programming [2]. The greatest interest was paid to this approach in connection with the problem of the shortest path in a graph [3, 4, 5, 6, 7], and this interest is yet alive as it follows from the recent discussion in electronic conferences. The discussion gave some new titles in this field, unfortunately I have no possibility to classify or even to list them now.

We consider here a general approach to enumeration of solutions rather than partial cases, and they serve us only as examples of possibilities and difficulties of our approach. Nevertheless, the application to nonserial dynamic programming should be mentioned, since I never met such considerations before.

Although several attempts were made to introduce a class of operations (see [9, 10], e.a.) the considered algorithms were always based on traditional forms of explicit representation of information, such as arrays. Abstraction of the data and a partial hiding of details allows us to simplify description of algorithms. Some of these operations are very close to ones considered. However, I never met there the operations allied to processes, and this distinction seems to be significant.

During my short visit to DIMACS, some experiments were made with software that can solve the simplest examples, its structure is described at the end of report to demonstrate the possibilities of implementation of our approach. Although it is yet far from ideal, it shows a possibility of efficient implementation of the idea of processes. A discussion held in the University of Texas, Austin, (see the end of the report), pointed me some links with new languages for functional programming, and particularly with Miranda.

I am grateful to V. Lifschitz and N. McCain, University of Texas, Austin, for the fruitful discussion, and to John Lee, MIT, who send me the draft of his paper on kth best paths in a graph.

A previous version of this paper (prepared with my student Ljubov' Skurina) is to appear in Russian. [8]

2 Enumeration process in an optimization problem

Consider a discrete optimization problem (f, A):

Let a function

$$f : A \to \mathcal{R}$$

be defined on a given finite set A. Find an element $x \in A$ that provided the minimum $f(x)$.

The difficulties in solving problems of this kind are well known. At the same time, in practice a solution of a single problem gives a negligible effect, because the computational difficulties affects us to simplify models, and the optimal solution of a simplified model does

ABSTRACT

We consider a new approach to enumeration of near-to-optimal solutions in discrete optimization problems. The main idea is to consider the processes that generate such solutions, and to describe possible operations with the processes.

It allows us to construct a rather standard tool for enumeration in those optimization problems that may be solved with ideas of splitting of the set of admissible solutions (as in Branch-and-Bound) and recursion (as in Dynamic programming).

As examples we consider the problems of shortest and the quickest paths in a graph, and the nonserial dynamic programming.

Some experimental software were developed to check the ideas. It was described at the end of the paper.

DIMACS Technical Report 93-64
October 1993

On Enumeration of Near to Best Solutions in Discrete and Dynamic Programming

by

Joseph V. Romanovsky[1]
Mathematical Faculty
St. Petersburg University
St. Petersburg, 198904, Russia

[1]Special Year Visitor to DIMACS, whose support is gratefully acknowledged

---

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.