**allocating INDIVISIBLE jobs in A MULTIPROCESSOR SYSTEM**

Khutoretskii A.B, Novosibirsk State University

Bredikhin S.V., Institute of Computational Mathematics and

Mathematical Geophysics, Novosibirsk

**1. Introduction.**

Assume that a system consisting of *n* processors provides paid services. There are *m* jobs, each of them can be included in а schedule for the planning period of duration *T*. Let *I* = {1, 2, …, *m*} be the set of jobs*’* numbers and
*J* = {1, 2, …, *n*} be the set of processors*’* numbers.

Each job can be assigned to any processor. Processor *j* has the speed *sj* (cycles per unit time) and capability *Qj* = *T* ∙ *sj* within the planning period. Job *i* requires *qi* processor cycles and has the value *ci*. If the job *i* is included in the schedule and will be, therefore, executed during the planning period, then the system obtains the payment *ci*.

Jobs are indivisible in two senses: (a) in the planning period, no part of a job is performed or this job terminates; (b) during the planning period, a job can use no more than one processor (scheduling without preemption). It follows from the jobs’ indivisibility that any schedule can be identified with some partition *I* = *I*0 ∪ *I*1 ∪ … ∪ *In*, where *i* ∈ *Ij* with *j* ∈ *J* iff the job *i* is allocated to the processor *j*, and the jobs with numbers in *I*0 are not executed.

A schedule is feasible if the total labor intensity of all jobs allocated to each processor is not greater than its capability. The more the total value of all jobs included in a schedule, the more preferable it is for the system.

Let *x* = (*xij* | *i* ∈ *I*, *j* ∈ *J*), where *xij* is the fraction of job *i* allocated to processor *j*. The problem of choosing the optimal schedule is formalized as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| maximize |  *V*(*x*) =  |  | (1) |
| subject to |   ≤ *Qj*, | *j* ∈ *J*, | (2) |
|   ≤ 1, | *i* ∈ *I*, | (3) |
|  *xij* ∈ {0, 1}, | *i* ∈ *I*, *j* ∈ *J*. | (4) |

Formally, this is the multiple knapsack problem (MKP), see, e. g. [4, section 10], [6, section 6]. Here processors correspond to knapsacks, size of the knapsack *j* equals to *Qj*, jobs correspond to items, the item *i*’s size and value are equal, respectively, to the laboriousness and value of the job *i*.

There exists one-to-one correspondence between feasible schedules and feasible solutions to the problem (1)-(4): if *j* ∈ *J*, then *i* ∈ *Ij* iff *xij* = 1, and *i* ∈ *I*0 iff *xij* = 0 for all *j* ∈ *J*.

**2. Related results and our contribution.**

The problem (1)-(4) is NP-hard [4, p. 298], so the approximate algorithms for solving MKP are of interest. Let us formulate the necessary definitions (see [4, p. 33-40]).

Let 𝒫 be some parametric optimization problem. Assume that an algorithm *A* generates a feasible solution *A*(*P*) for any instance *P* of 𝒫. The relative performance guarantee or approximation ratio of the algorithm *A* for 𝒫 is the greatest ε such that *A*(*P*) ≥ εOpt(*P*) for all instances *P* of 𝒫 (here Opt(*P*) is the optimal value of the objective function in the problem *P*). An algorithm with relative performance guarantee ε will be called an ε-approximate algorithm.

A dependent on ε algorithm *A* is anapproximation scheme for the problem 𝒫 if it is (1 – ε)-approximate algorithm for anyε ∈ (0, 1). Anapproximation scheme *A* is apolynomial time approximation scheme (PTAS) if its running time is polynomial in the parameters of the problem 𝒫*.* An approximation scheme *A* is a fully polynomial time approximation scheme (FPTAS) if its running time is polynomial in the parameters of the problem 𝒫 and in 1 / ε.

In [1], it was proved that if *P* ≠ *NP* then there is no FPTAS for MKP even with *n* = 2. A PTAS for MKP was also constructed there, which allows to obtain a (1 – ε)-approximate solution with the running time of  (this estimate of the running time was given in [3]).

Looking through the pairs (*i*; *j*) ∈ *I* × *J* in some order and placing the item *i* in the knapsack *j* if the item has not yet been placed and there is enough space in the knapsack, it is possible to construct a feasible solution of MKP. Different orderings of pairs generate a family of greedy algorithms for MKP; we call them packing. The definition of packing can be easily modified for problems with divisible items (for example, the linear relaxation of MKP). Namely, if (*i*; *j*) is the next in turn pair,  is the size of item *i*’s unplaced part, and  is he size of free space in the knapsack *j*, then let us pack into this knapsack the item *i*’s portion of size 

MKP with *n* = 1 is the knapsack problem (KP). Assuming that the size of every item is not larger than the knapsack’s size, an optimal solution *z*KP for the linear relaxation of KP can be obtained by packing with the numbering of items according to non-increase of the efficiency *di* = *ci* / *qi* [2].

Let *z*KP = (*zij* | *i* ∈ *I*, *j* ∈ *J*   ). One can construct two feasible solutions to KP, *x*KP = (*xij* | *i* ∈ *I*, *j* ∈ *J*   ) and *y*KP = (  *yij* | *i* ∈ *I*, *j* ∈ *J*   ), as follows: *xij* = 1 if *zij* = 1, otherwise *xij* = 0; *yij* = 1 if 0 < *zij* < 1, otherwise *zij* = 0. Algorithm KP1 that chooses the best of these solutions is 0.5-approximate one for KP [6].

For MKP, the analogous approach was described in [4, p. 299]. Here the linear relaxation of MKP is considered with an additional condition: it is forbidden to place a nonzero part of an item *i* in a knapsack *j* if *Qj* *< qi* (we call this problem LMKP). If the items are numbered by non-increasing the efficiency and the knapsacks are numbered by non-decreasing the size, then the packing that uses only the pairs (*i; j*) such that *qi* ≤ *Qj* (in a lexicographical order) creates an optimal solution to LMKP (a more general result is proved in [5, Theorem 1]). Let *z*MKP = (*zij* | *i* ∈ *I*, *j* ∈ *J*   ) be the optimal solution of LMKP constructed by the method described above. This vector creates two feasible solutions of MKP,
*x* = (*xij* | *i* ∈ *I*, *j* ∈ *J*   ) and *y* = (  *yij* | *i* ∈ *I*, *j* ∈ *J*   ), as follows: *xij* = 1 if *zij* = 1, otherwise *xij* = 0; *yij* = 1 if 0 < *zij* < 1 and *zik* = 0 for all *k* < *j*, otherwise *yij* = 0. Denote as MKP1 the algorithm that chooses the best of these solutions. This algoritm is 0.5-approximate one for MKP [4, Theorem 10.4.2]. Running time of the algorithm (without sorting) is *O*(*mn*).

In [7], the authors suggest to improve the algorithm KP1 as follows: to establish the items’ numbering according to non-increasing the value and use the result of the corresponding packing instead of *y*KP. In other words, the improved algorithm KP2 performs the packing with the items’ numbering by non-increasing the efficiency and one more packing – with numbering of items by non-increasing the value, then chooses the best from the received solutions. Estimates of the running time and approximation ratio for KP2 coincide with the corresponding characteristics of the algorithm KP1.

We apply the idea of the algorithm KP2 to construct a new 0.5-approximate algorithm for MKP, having the running time of *O*(*mn*). This may be useful because, for a particular instance of MKP, different algorithms give, as a rule, different solutions from which one can choose the best.

**3. The proposed algorithm.**

Before description of the proposed algorithm (MKP2 below), let us first formulate a greedy algorithm (packing) for distributing the indivisible jobs between processors using given numberings of the jobs and processors.

**Packing**

*Input*: the sets *I*, *J*, {*qi* | *i* ∈ *I* }, {*Qj* | *j* ∈ *J* }.

*Step* 1. Put *I*0 = *I* and *Ij* = ∅ for all *j* ∈ *J*.

For each *j* from 1 to *n* execute the step 2.

*Step* 2. For each *i* from 1 to *m* do

put *Rj* = *Qj* – 

if *i* ∈ *I*0 and *qi* ≤ *Rj* then relocate *i* from *I*0 into *Ij*.

*Output*: the partition ρ = (*I*0, *I*1, … *In*) of the set *I*.

The proposed 0.5-effective algorithm MKP2 consists in double execution of the packing. In the both cases the processors are numbered according to non-decreasing the size. The jobs are numbered according to non-increasing the efficiency in the first case and according to non-increasing the value in the second case.

**Algorithm MKP2**

*Input*: the sets *I*, *J*, {*qi* | *i* ∈ *I* }, {*ci* | *i* ∈ *I* }, and {*Qj* | *j* ∈ *J* }, where *j* < *k* implies *Qj* ≤ *Qk*.

*Step* 1. Establish a numbering of the set *I* such that *di* ≥ *dk* for *i* < *k*.

Performing the packing, find a partition ρ1 = (*I*1,0, *I*1,1, … *I*1,*n*) of the set *I*.

Put *V*1 = 

*Step* 2. Establish a numbering of the set *I* such that *ci* ≥ *ck* for *i* < *k*.

Performing the packing, find a partition ρ2 = (*I*2,0, *I*2,1, … *I*2,*n*) of the set *I*.

Put *V*2 = 

*Step* 3. Put ρ = ρ1 if *V*1 ≥ *V*2 and ρ = ρ2 otherwise.

*Output*: the partition ρ of the set *I*.

Obviously, the algorithm MKP2 has the running time of *O*(*mn*) (without sorting). Using the resulting partition ρ = (*I*0, *I*1, … *In*), it is possible to construct a feasible solution *x* = (*xij* | *i* ∈ *I*, *j* ∈ *J*   ) of the MKP as follows: *xij* = 1 if *i* ∈ *Ij*, else *xij* = 0. Therefore, MKP2 is an approximate algorithm for the MKP. In [5], it is proved (Theorem 3) that 0.5 is a lower bound for the approximation ratio of this algorithm. In other words, for arbitrary input parameters, the algorithm includes in the schedule the jobs of total value not less than half of the value of jobs in the optimal schedule.

Example 1 below shows that this estimate is tight even with *n* = 1.

Let *P* be an instance of MKP. Algorithm MKP*k*, *k* ∈ {1, 2}, creates two partitions, ρ*ks*, where *s* ∈ {1, 2}. Let *Vks* be the total value of scheduled jobs in the corresponding schedules. Then *Vk*(*P*) = max{*Vk*1, *Vk*2} is the resulting value produced by the algorithm MKP*k*. Examples 2 and 3 below show that, dependent on *P*, both cases *V*1(*P*) > *V*2(*P*) and *V*2(*P*) > *V*1(*P*) are possible.

**References**

1. Chekuri, С. A polynomial time approximation scheme for the multiple knapsack problem / C. Chekuri, S. Khanna // SIAM Journal on Computing. – 2003. – Vol. 5. – Is. 3. – P. 713-728.

2. Dantzig, G. D. Discrete variable extremum problems / G. D. Dantzig // Operations Research. – 1957. – Vol. 5. – P. 266-277.

3. Jansen, K. Parameterized approximation scheme for the multiple knapsack problem / K. Jansen // SIAM Journal on Computing. – 2009. Vol. 39. – Is. 4. – P. 1392-1412.

4. Kellerer, H. Knapsack problems / H. Kellerer, U. Pferschy, D. Pisinger. – Berlin, Heidelberg: Springer, 2004. – 546 p.

5. Khutoretskii, A. B. A lexicographic 0:5-approximation algorithm for the multiple knapsack problem / A. B. Khutoretskii, S.V. Bredikhin, A.A. Zamyatin // Journal of Applied and Industrial Mathematics. – 2018. – Vol. 12. – [Is. 2](https://link.springer.com/journal/11754/12/2/page/1). – P. 264-277.

6. Martello, S.. Knapsack problems: algorithms and computer implementations / S. Martello, P Toth. – Chichester – New York: John Wiley & Sons, 1990. – 296 p.

7. Mu’alem, A. Truthful approximation mechanisms for restricted combinatorial auctions / A. Mu’alem, N. Nisan // Games and Economic Behavior. – 2008. – Vol. 64. – P. 612-631.

**Example 1**:  = 0.5. One processor, *Q* = 6.

|  |  |  |
| --- | --- | --- |
| Parameters of jobs |  | optimal partition, *V*   \* = 6 |
| job | *q* | *c* | *d* |  | 2 | 2 | 2 |  | 3 | 3 | 3 |
| 1 | 3 + ε | 3 + 2ε | > 1 |  | heuristic partition, *V*(ε) = 3 + 2ε |
| 2 | 3 | 3 | 1 |  | 1 | 1 | 1 | 1 |  |  |  |
| 3 | 3 | 3 | 1 |  |  |  |  | ε |  |  |  |

**Example 2**: An instance *P* of MKP such that the algorithm MKP1 collects a larger value than MKP2. Two processors, *Q*1 = *Q*2 = 8.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| job | 1 | 2 | 3 | 4 | 5 |
| *q* | 3 | 3 | 3 | 6 | 7 |
| *c* | 5 | 5 | 4 | 6 | 6 |
| *d* | 5 / 3 | 5 / 3 | 4 / 3 | 1 | 6 / 7 |

Algorithm MKP1 constructs partitions ρ1,1, ρ1,2 and chooses ρ1,1 with value *V*1(*P*) = 16.

 partition ρ1,1, *V*1,1 = 16 partition ρ1,2, *V*1,2 = 10

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 2 | 2 | 2 |   |   | processor 1 | 3  | 3  | 3  |   |   |   |   |   |
|   | 4 | 4 | 4 | 4 | 4 | 4 |  | processor 2 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |  |

Algorithm MKP2 constructs partitions ρ2,1, ρ2,2 and chooses ρ2,1 with value *V*2(*P*) = 14.

 partition ρ2,1, *V*2,1 = 14 partition ρ2,2, *V*2,2 = 12

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 2 | 2 | 2 |   |   | processor 1 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |   |
| 3  | 3  | 3  |  |  |  |  |  | processor 2 | 4 | 4 | 4 | 4 | 4 | 4 |  |  |

So, *V*1(*P*) > *V*2(*P*).

**Example 3**: An instance *P* of MKP such that the algorithm MKP2 collects a larger value than MKP1. Two processors, *Q*1 = *Q*2 = 8.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| job | 1 | 2 | 3 | 4 |
| *q* | 3 | 5 | 2 | 4 |
| *c* | 6 | 8 | 3 | 5 |
| *d* | 2 | 1.6 | 1.5 | 1.2 |

Algorithm MKP1 constructs partitions ρ1,1, ρ1,2 and chooses ρ1,2 with value *V*1(*P*) = 13.

 partition ρ1,1, *V*1,1 = 9 partition ρ1,2, *V*1,2 = 13

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 |   |   |   | processor 1 | 2 | 2 | 2 | 2 | 2 |  |
|  |  | 3 | 3 |  |  | processor 2 | 4 | 4 | 4 | 4 |  |  |

Algorithm MKP2 constructs partitions ρ2,1, ρ2,2.

 partition ρ2,1, *V*2,1 = 17 partition ρ2,2, *V*2,2 = 17

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 3 | 3 |   | processor 1 | 2 | 2 | 2 | 2 | 2 |  |
| 2 | 2 | 2 | 2 | 2 |  | processor 2 | 1 | 1 | 1 | 3 | 3 |  |

So, *V*2(*P*) = 17 and *V*2(*P*) > *V*1(*P*).