# Graphical Algorithm for the Knapsack Problems

Alexander Lazarev, Anton Salnikov, and Anton Baranov

Institute of Control Sciences of the Russian Academy of Sciences,
Profsoyuznaya st. 65, 117997 Moscow, Russia,
Lomonosov Moscow State University,
State University – Higher School of Economics,
Moscow Institute of Physics and Technology – State University
jobmath@mail.ru,salnikov@ipu.ru,
av.baranov@physics.msu.ru

**Abstract.** We consider a modification of dynamic programming algorithm ($DPA$), which is called as graphical algorithm ($GA$). For the knapsack problem ($KP$) it is shown that the time complexity of $GA$ is less than the time complexity of $DPA$. Moreover, the running time of $GA$ is often essentially reduced. $GA$ can also solve big scale instances and instances, where the parameters are not only positive integer. The paper outlines different methods of parallelizing $GA$ taking into account its main features and advantages to various parallel architectures, in particular by using $OpenCL$ and $MPI$ framework. Experiments show that "hard" instances of $KP$ for $GA$ have correlation $p_j \simeq kw_j$ for all $j$, where $p_j$ and $w_j$ are utility and capacity of item $j = 1, 2, \ldots, n$.

**Keywords:** Graphical Algorithm, Knapsack Problem, OpenCL, MPI, Parallel Algorithm.

## 1 Introduction

Dynamic programming is a general optimization technique developed by Bellman [1]. It can be considered as a recursive optimization procedure which interprets the optimization problem as a multi-step solution process. Bellman's optimality principle can be briefly formulated as follows: Starting from any current step, an optimal policy for the subsequent steps is independent of the policy adopted in the previous steps. In the case of a combinatorial problem, at some step $j$, $j = 2, \ldots, n$, sets of a particular size $j$ are considered. To determine the optimal criterion value for a particular subset of size $j$, one has to know the optimal values for all necessary subsets of size $j - 1$. If the problem includes $n$ elements, the number of subsets to be considered is equal to $O(2^n)$. Therefore, dynamic programming usually results in an exponential complexity. However, if the problem considered is $NP$-hard in the ordinary sense, it is possible to derive pseudo-polynomial algorithms [2,3,4].

In this paper, we give the basic idea of a graphical modification of dynamic programming algorithm ($DPA$), which is called Graphical Algorithm ($GA$). This approach often reduces the number of it states to be considered in each step of

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 459–466, 2011.

a $DPA$. Moreover, in contrast to classical $DPA$, it can also treat problems with non-integer data without necessary transformations of the corresponding problem. In addition, for some problems, $GA$ essentially reduces the time complexity.

For the knapsack problem DPA with the same idea like in $GA$ are known (e.g. see [7]). In such $DPA$ not all states $t \in [0, C]$ are considered, but only states, where a value of objective function is changed. Thus, the time complexity of such $DPA$ is bounded by $O(nF_{opt})$, where $F_{opt}$ is the optimal value of objective function. However, these algorithms can be useful only for problems, where $F_{opt} < C$, otherwise we can use the classical $DPA$. We generalize the idea of such algorithms for the objective function, for which $F_{opt} \gg C$.

This paper is organized as follows. In Section 2, we give the basic idea of the $GA$. In the next section we describe graphical algorithm for the binary knapsack problem. Section 4 describes parallel implementation of $GA$ using $OpenCL$ and $MPI$ framework. Last section represents the results of experiments to search for and analyse of "hard" examples.

## 2 Basic Idea of the Graphical Algorithm

Usually in $DPA$, we have to compute the value $f_j(t)$ of a particular function for each possible state $t$ at each stage $j$ of a decision process, where $t \in [0, C]$ and $t, C \in Z^+$. If this is done for any stage $j = 1, 2, \ldots, n$, where $n$ is a size of the problem, the time complexity of such a $DPA$ is typically $O(nC)$. However, often it is not necessary to store the result for any integer state since in the interval $[t_l, t_{l+1})$, we have a functional equation $f_j(t) = \varphi(t)$ (e.g. $f_j(t) = k_j \cdot t + b_j$, i.e., $f_j(t)$ a continuous linear function when allowing also real values $t$).

Assume that we have the following functional equations in a $DPA$, which correspond to Bellman's recursive equations:

$$f_j(t) = \min_{j=1,2,\ldots,n} \begin{cases} \Phi^1(t) = \alpha_j(t) + f_{j-1}(t - w_j) \\ \Phi^2(t) = \beta_j(t) + f_{j-1}(t - b_j) \end{cases} \tag{1}$$

with the initial conditions

$$\begin{array}{ll} f_0(t) = 0, & \text{for } t \geq 0, \\ f_0(t) = +\infty, & \text{for } t < 0. \end{array} \tag{2}$$

In (1), function $\Phi^1(t)$ characterizes a setting $x_j = 1$ while $\Phi^2(t)$ characterizes a setting $x_j = 0$ representing a yes/no decision, e.g. for an item, a job [2],[6]. In step $j$, $j = 1, 2, \ldots, n$, we compute and store the data given in Table 1.

Here $X(y), y = 0, 1, \ldots, C$, is a vector which describes an optimal partial solution and which consists of $j$ elements (values) $x_1, x_2, \ldots, x_j \in \{0, 1\}$.

**Table 1.** Computations in $DPA$

| $t$ | 0 | 1 | 2 | ... | $y$ | ... | $C$ |
|---|---|---|---|---|---|---|---|
| $f_j(t)$ | $value_0$ | $value_1$ | $value_2$ | ... | $value_y$ | ... | $value_C$ |
| optimal partial solution $X(t)$ | $X(0)$ | $X(1)$ | $X(2)$ | ... | $X(y)$ | ... | $X(C)$ |

**Table 2.** Computations in $GA$

| $t$ | $[t_0, t_1)$ | $[t_1, t_2)$ | $\ldots$ | $[t_l, t_{l+1})$ | $\ldots$ | $[t_{m_j-1}, t_{m_j}]$ |
|---|---|---|---|---|---|---|
| $f_j(t)$ | $\varphi_1(t)$ | $\varphi_2(t)$ | $\ldots$ | $\varphi_{l+1}(t)$ | $\ldots$ | $\varphi_{m_j}(t)$ |
| optimal partial solution $X(t)$ | $X(t_0)$ | $X(t_1)$ | $\ldots$ | $X(t_l)$ | $\ldots$ | $X(t_{m_j-1})$ |

However, this data can also be stored in a condensed tabular form as given in Table 2.

Here, we have $0 = t_0 < t_1 < t_2 < \ldots < t_{m_j} = C$.

To compute function $f_{j+1}(t)$, we compare two temporary functions $\Phi^1(t)$ and $\Phi^2(t)$.

The function $\Phi^1(t)$ is a combination of the terms $\alpha_{j+1}(t)$ and $f_j(t - w_{j+1})$. Function $f_j(t - w_{j+1})$ has the same structure as in Table 2, but all intervals $[t_l, t_{l+1})$ have been replaced by $[t_l - w_{j+1}, t_{l+1} - w_{j+1})$, i.e., we shift the graph of function $f_j(t)$ to the right by the value $w_{j+1}$. If we can present function $\alpha_{j+1}(t)$ in the same form as in Table 2 with $\mu_1$ columns, we store function $\Phi^1(t)$ in the form of Table 2 with $m_j + \mu_1$ columns. In an analogous way, we store function $\Phi^2(t)$ in the form of Table 2 with $m_j + \mu_2$ columns.

Then we construct function

$$f_{j+1}(t) = \min\{\Phi^1(t), \Phi^2(t)\}.$$

For example, let the columns of Table $\Phi^1(t)$ contain the intervals

$$[t_0^1, t_1^1), [t_1^1, t_2^1), \ldots, [t_{(m_j+\mu_1)-1}^1, t_{(m_j+\mu_1)}^1]$$

and the columns of Table $\Phi^2(t)$ contain the intervals

$$[t_0^2, t_1^2), [t_1^2, t_2^2), \ldots, [t_{(m_j+\mu_2)-1}^2, t_{(m_j+\mu_2)}^2].$$

To construct function $f_{j+1}(t)$, we compare the two functions $\Phi^1(t)$ and $\Phi^2(t)$ on each interval, which is formed by means of the points

$$\{\, t_0^1, t_1^1, t_2^1, \ldots, t_{(m_j+\mu_1)-1}^1, t_{(m_j+\mu_1)}^1, \\ t_0^2, t_1^2, t_2^2, \ldots, t_{(m_j+\mu_2)-1}^2, t_{(m_j+\mu_2)}^2 \},$$

and we determine the intersection points $t_1^3, t_2^3, \ldots, t_{\mu_3}^3$. Thus, in the table of function $f_{j+1}(t)$, we have at most $2m_j + \mu_1 + \mu_2 + \mu_3 \leq C$ intervals.

In fact, in each step $j = 1, 2, \ldots, n$, we do not consider all points $t \in [0, C]$, $t$, $C \in Z^+$, but only points from the interval in which the optimal partial solution changes or where the resulting functional equation of the objective function changes. For some objective functions, the number of such points $M$ is small and the new algorithm based on this graphical approach has a time complexity of $O(n \min\{C, M\})$ instead of $O(nC)$ for the original $DPA$.

Moreover, such an approach has some other advantages.

1. The $GA$ can solve instances, where $p_j$, $w_j$, $j = 1, 2, \ldots, n$, or/and $C$ are not integer.

2. The running time of the $GA$ for two instances with the parameters $\{p_j, w_j, C\}$ and $\{p_j \cdot 10^\alpha \pm 1,\ w_j \cdot 10^\alpha \pm 1, C \cdot 10^\alpha \pm 1\}$ is the same while the running time of the $DPA$ will be $10^\alpha$ times larger in the second case. Thus, using the $GA$, one can usually solve considerably larger instances.
3. Properties of an optimal solution are taken into account. For KP, an item with the smallest value $\frac{p_j}{w_j}$ may not influence the running time.
4. As we will show below, for several problems, $GA$ has even a polynomial time complexity or we can at least essentially reduce the complexity of the standard $DPA$.

Thus, the use of $GA$ can reduce both the time complexity and the running time for $KP$. The application of $GA$ to the partition problem is described in detail in [5], where also computational results are presented.

## 3 Graphical Algorithm for the Knapsack Problem

In this section, we describe the application of this approach to the one-dimensional knapsack problem [5].

**One-dimensional knapsack problem ($KP$):** One wishes to fill a knapsack of capacity $C$ with items having the largest possible total utility. If any item can be put at most once into the knapsack, we get the binary or $0-1$ knapsack problem. This problem can be written as the following integer linear programming problem:

$$\begin{cases} f(x) = \sum\limits_{j=1}^{n} p_j x_j \to \max \\ \sum\limits_{j=1}^{n} w_j x_j \le C, \\ x_j \in \{0,1\},\ j = 1,2,\ldots,n. \end{cases} \tag{3}$$

Here, $p_j$ gives the utility and $w_j$ the required capacity of item $j$, $j = 1,2,\ldots,n$. The variable $x_j$ characterizes whether item $j$ is put into the knapsack or not.

The $DPA$ based on Bellman's optimality principle is one of the standard algorithms for the $KP$. It is assumed that all parameters are positive integer: $C, p_j, w_j \in Z^+, j = 1,2,\ldots,n$.

For $KP$, Bellman's recursive equations are as follows:

$$f_j(t) = \max_{j=1,2,\ldots,n} \begin{cases} \Phi^1(t) = p_j + f_{j-1}(t - w_j) \\ \Phi^2(t) = f_{j-1}(t), \end{cases} \tag{4}$$

where

$$f_0(t) = 0, \quad t \ge 0,$$
$$f_0(t) = +\infty,\ t < 0.$$

$\Phi^1(t)$ represents the setting $x_j = 1$ (i.e., item $j$ is put into the knapsack) while $\Phi^2(t)$ represents the setting $x_j = 0$ (i.e., item $j$ is not put into the knapsack). In each step $j$, $j = 1,2,\ldots,n$, the function values $f_j(t)$ are calculated for each integer point (i.e., "state") $0 \le t \le C$. For each point $t$, a corresponding best (partial) solution $X(t) = (x_1(t), x_2(t), \ldots, x_j(t))$ is stored.

## 4 Parallel Implementation

This section describes the parallel implementation of $GA$ using $OpenCL$ framework and $MPI$.

### 4.1 Column Parallelization

The classical Bellman's recurrence (4) can be implemented by creating an array with $C + 1$ rows and $n$ columns to store all the values in each step of the program that sequentially adds items into the problem by filling up the table column by column, where $C$ – is capacity of the knapsack. To compute the values for any column the values from the previous column are only needed. Such economical consumption of memory allows us easily implement this algorithm using a variety of parallel programming models including shared memory and distributed memory $API$s.

The specificity of $GA$ allows us to renounce the use of the columns of the same length $C$ in each step of the host program, and gradually increase the number of parallel processors while we add items to the problem. For the $GA$ "length" of the column in each step depends on the distance between the boundary break points. Furthermore, taking into account this feature we could essentially reduce the running time of the program by initial sorting the items in non-decreasing order of values $w_j$.

Fig. 1, 2 show plots of the number of break-points for $GA$ and $DPA$ in the two cases: when $C$ is chosen at a rate of 10% and 90% of the $C_{\max} = \sum_{i=1}^{n} w_i$.

### 4.2 Interval Parallelization

This parallelization is not easy to implement but it allows us to take main advantage of the $GA$.

When we look closer to the $GA$ we note that to create a table of intervals in step $j$, $j = 2, \ldots, n$, of the sequential program all we need are values from the

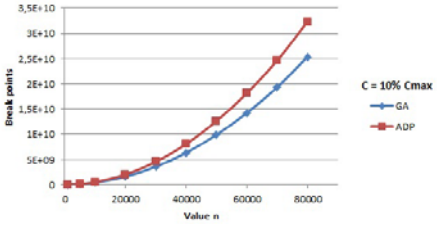


**Fig. 1.** The dependence of the number of break points on the number of items. $C = 10\%C_{\max}$.

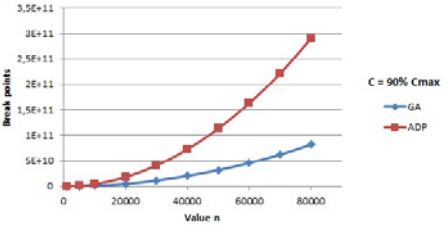


**Fig. 2.** The dependence of the number of break points on the number of items. $C = 90\%C_{\max}$.

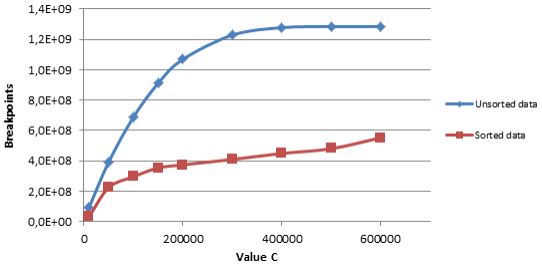


**Fig. 3.** The dependence of the number of break-points on the value of $C$ for sorted and unsorted initial data

previous table in step $j-1$. Therefore in each step we can partition the table of intervals from the previous step into columns, and make each parallel processor responsible for only one column i.e. one interval between two break points. Then each processor has to update information about break points and corresponding intervals for the table in the current step.

This is quite a tricky procedure because we could obtain one, two or none current intervals depending on situation for each previous interval. Furthermore, every parallel processor should have access to the table of intervals to calculate the new values. The last obstacle is easily bypassed on shared memory models but the first one requires additional memory and computational resources to store and extract the temporal information given by every parallel processor for the current table of intervals in each step of the sequential program. The practical realization of this procedure depends on the chosen data structure.

When parallelizing intervals we also should note that in each step of the sequential program the number of intervals may be doubled, so the partition of the table of intervals in a coarse-grained manner without regular reassigning columns to the new parallel processors could lead to the explosive load on one or several parallel processors and full stop of the program.

While solving large instances it is helpful to control the load of the grid by initial sorting the items in non-increasing order of values $\frac{p_j}{w_j}$. Fig. 3 shows the plots for examples with dimension $n = 10000$.

## 5 Experiments

The main objective of experiments was to search and analyse "hard" examples for which graphical algorithm would show the maximal time complexity. The basic unit of $GA$ is the break point. More break points we have are for more complex problem we encounter. Thus, the main objective of our programming activities was to create the procedure for finding instances with so many break points as possible.

It is obvious that the theoretical maximum number of break points is $2^{n+1}-1$. To obtain this result in real example on positive integers the value of $C$ should

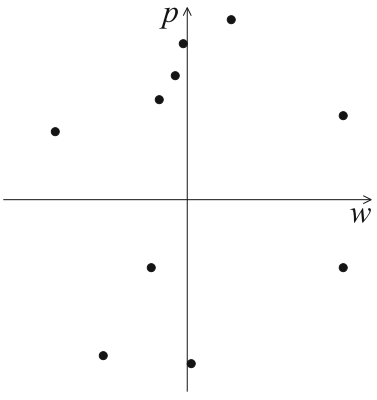


**Fig. 4.** Initial instance, n=10, B=37

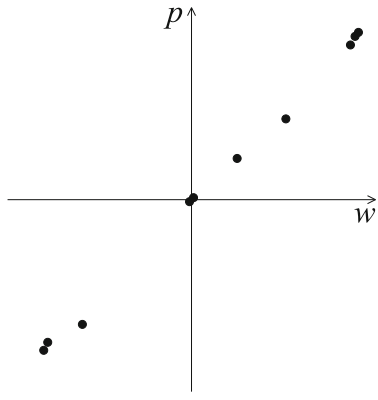


**Fig. 5.** Hard instance, n=10, B=2047

be more than maximum number of break points or we should simply exclude $C$ and associated restrictions from the problem. Additionally, to study the $GA$ deeply we should totally switch into integers by allowing negative values for $w_j$ and $p_j$.

As mentioned before the maximum number of break points that we can get is $2^{n+1} - 1$. This is the most complicated example that is practically impossible to get randomly as it turned out during experiment which instead of it was giving us less than $n^2$ break points for the most of the times. After a series of experiments we came up to some heuristic procedure that could increase the number of break points through the gradual change of the parameters of the initial randomly generated instances.

Our method, despite its apparent simplicity, has shown to be highly effective in the rapid searching for hard instances for which the number of break points is approaching to the maximum value $2^{n+1} - 1$. Fig. 4 and Fig. 5 show first and last stages of this process correspondingly. The experiment was carried out for $n = 10$, $w_j$ and $p_j$ are integers drawn from the normal distribution within interval [-1024, 1024], and $B$ is number of break points.

It is easy to look that all points in Fig. 5 lie very close to the line passing through the center of coordinates $(0, 0)$. This property became apparent in all our experiments, which gives us the right to predicate that all hard instances of $KP$ for $GA$ should satisfy the following correlation $p_j \simeq kw_j$, $j = 1, \ldots, n$.

In summary the most complex instance of $KP$ for $GA$ can be written as follows:

$$\begin{cases} \sum\limits_{j=1}^{n} k \cdot w_j x_j \to \max \\ \sum\limits_{j=1}^{n} w_j x_j \le C, \\ x_j \in \{0, 1\},\ j = 1, 2, \ldots, n. \end{cases} \tag{5}$$

## 6 Concluding Remarks

The graphical approach can be applied to problems where a pseudo-polynomial algorithm exists and Boolean variables are used in the sense that yes/no decisions have to made (e.g. in the problem under consideration, for $KP$, an item can be put into the knapsack or not), for example for partition and scheduling problems. However, for the knapsack problem, the graphical algorithm mostly reduces substantially the number of points to be considered but the time complexity of the algorithm remains pseudo-polynomial.

**Acknowledgements.** Partially supported by programs of Russian Academy of Sciences 15 and 29. We would like to thank Prof. Dr. Frank Werner and Dr. Evgeny Gafarov for many discussions and helpful suggestions.

## References

1. Bellman, R.: Dynamic Programming. Princeton Univ. Press, Princeton (1957)
2. Gafarov, E.R., Lazarev, A.A., Werner, F.: Algorithms for Some Maximization Scheduling Problems on a Single Machine. Automation and Remote Control 71(10), 2070–2084 (2010)
3. Keller, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Hidelberg (2010)
4. Lawler, E.L., Moore, J.M.: A Functional Equation and its Application to Resource Allocation and Sequencing Problems. Management Science 16(1), 77–84 (1969)
5. Lazarev, A.A., Werner, F.: A Graphical Realization of the Dynamic Programming Method for Solving $NP$-Hard Combinatorial Problems. Computers and Mathematics with Applications 58(4), 619–631 (2009)
6. Lazarev, A.A., Werner, F.: Algorithms for Special Cases of the Single Machine Total Tardiness Problem and an Application to the Even-Odd Partition Problem. Mathematical and Computer Modelling 49(9-10), 2061–2072 (2009)
7. Papadimitrou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, INC., New York (1998)