The Ninth International Symposium on Operations Research and Its Applications (ISORA'10)
Chengdu-Jiuzhaigou, China, August 19–23, 2010
Copyright © 2010 ORSC & APORC, pp. 144–151

Shift-and-Merge Technique for the DP Solution of the Backpacker Problem

Byungjun You Takeo Yamada

Department of Computer Science, The National Defense Academy
Yokosuka, Kanagawa 239-8686, Japan

Abstract We formulate the 'backpacker problem' as an extension of the binary knapsack problem, and present two dynamic programming (DP) algorithms to solve this problem to optimality. Firstly, we propose a DP algorithm that solves the problem in pseudo polynomial time. Based on the observation that the optimal objective function is a step function of the knapsack capacity, we present an improved, 'shift-and-merge' DP algorithm. Although this technique does not change the worst case complexity, in practical computation the computing time is significantly reduced.

We implement the DP and shift-and-merge DP algorithms in ANSI-C language, and evaluate the performance of these algorithms for various type and size of instances, including comparison against MIP solvers.

Keywords Combinatorial optimization, Dynamic programming, Knapsack problem, Directed acyclic graph.

1 Introduction

We are concerned with a variation of the standard binary *knapsack problem* (KP, [9, 10]), where a 'backpacker' travels from a origin to a destination on a *directed acyclic graph* (DAG, [13]), and collects items *en route* within the capacity of his knapsack. To formulate this problem, let $G = (V, E)$ be a DAG with node set $V = \{v_1, v_2, \cdots, v_n\}$ and arc set $E = \{e_1, e_2, \cdots, e_m\} \subseteq V \times V$. Node v_1 is the origin, and v_n is the destination, and we assume that there exists at least one path from v_1 to v_n. Each node $v_j \in V$ is associated with an item of weight w_j and profit p_j, and we sometimes call this item j. The capacity of the backpacker's knapsack is B.

We prepare some graph notations [1]. For $e = (v, v') \in E$, we write $\partial^- e = v$ and $\partial^+ e = v'$, and for $v \in V$ define the sets of incoming and outgoing arcs as $E^-(v) = \{e \in E \mid \partial^+ e = v\}$ and $E^+(v) = \{e \in E \mid \partial^- e = v\}$, respectively. Let us introduce decision variables as follows: $x_j = 1$ if the backpacker accepts item j, and $x_j = 0$ otherwise. Similarly, $y_e = 1$, if he takes a path that includes arc e, and $y_e = 0$ otherwise.

Then, the *backpacker problem* (BP) is formulated mathematically as follows.

BP:

$$\text{Maximize} \quad \sum_{j \in V} p_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j \in V} w_j x_j \leq B, \tag{2}$$

$$\sum_{e \in E^+(v_j)} y_e - \sum_{e \in E^-(v_j)} y_e = \begin{cases} 1, & \text{if } j = 1 \\ -1, & \text{if } j = n, \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$x_j \leq \sum_{e \in E^-(v_j)} y_e, \quad \forall j \in V \setminus \{1\}, \tag{4}$$

$$x_j, y_e \in \{0, 1\}, \quad \forall j \in V, \forall e \in E. \tag{5}$$

Here, (1) and (2) are as in ordinary KP, and (3) and (5) determines a 0-1 vector that corresponds to a path from v_1 to v_n. Inequality (4) means that only items on the path can be accepted.

Without much loss of generality, we assume that problem data w_j, p_j ($j = 1, 2, \cdots, n$) and B are all positive integers, and $w_j \leq B$ ($\forall j \in V$) and $\sum_{j \in V} w_j > B$, since otherwise the problem is trivial. BP is $\mathcal{NP}$-hard, since it includes KP which is already $\mathcal{NP}$-hard [7].

Remark 1 Node $v_i \in V$ is *maximal* if $E^-(v_i) = \emptyset$, and *minimal* if $E^+(v_i) = \emptyset$. We assume that v_1 is the only one maximal node, and v_n is the only one minimal node. That is, for all intermediate nodes we have $E^{\pm}(v_i) \neq \emptyset$ ($i = 2, \cdots, n-1$), since otherwise no path exists from v_1 to v_n via v_i. □

Remark 2 It is possible to consider BP on a directed graph in general, where the graph may have some directed cycles. However, we can reduce such a problem to BP on a DAG by transforming the graph as follows. If G is not a DAG, its nodes can be classified into a set of *strongly connected components* [1]. Then, if $C = \{v^1, v^2, \cdots, v^p\}$ is such a component, we modify G into a DAG G' as follows. Instead of arcs connecting nodes in C, we prepare a series of arcs connecting $v^1 \to v^2 \to \cdots \to v^p$. Next, we make all the arcs entering into (exiting from) C now enter into (exit from) node v^1 (v^p), respectively. Thus we obtain BP on a DAG. □

Knapsack problem on a directed graph has been studied as a precedence-constrained knapsack problem [12, 15], or a tree knapsack problem [3, 8]. However, in the backpacker problem we need to determine the set of items to be accepted, as well as path from v_1 to v_n. To our knowledge, no previous literature treated these two aspects simultaneously.

Since BP is a linear 0-1 programming problem, small instances may be solved using mixed integer programming (MIP, [14]) solvers such as CPLEX [5] or XPRESS-MP. For larger instances, however, it is often difficult to obtain exact solutions by such an approach. In this article, we present two *dynamic programming* (DP, [2]) algorithms for such an instance. In Section 2, we present a DP algorithm to solve BP on a DAG. This is improved in Section 3 to a DP algorithm with *lists* [6, 11], enabling us to solve instances with up to 320000 items within a few seconds in ordinary computing environment.

2 A dynamic programming algorithm

We consider BP on a DAG $G = (V,E)$. Without loss of generality, the nodes are assumed to be *topologically sorted* [4, 13], in the sense that $(v_i, v_j) \in E \Leftrightarrow i < j$. For node $v_i \in V$, we introduce $G_i = (V_i, E_i)$ as the subgraph of G restricted to the downstream of node v_i, i.e., $V_i = \{v_j \in V \mid j \geq i\}$ and $E_i = \{e \in E \mid \partial^{\pm} e \geq i\}$, and define a subproblem of BP on G_i as follows.

BP$_i(b)$:

$$\text{Maximize} \quad \sum_{j \in V_i} p_j x_j \tag{6}$$

$$\text{subject to} \quad \sum_{j \in V_i} w_j x_j \leq b, \tag{7}$$

$$\sum_{e \in E^+(v_j)} y_e - \sum_{e \in E^-(v_j)} y_e = \begin{cases} 1, & \text{if } j = i \\ -1, & \text{if } j = n, \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

$$x_j \leq \sum_{e \in E^-(v_j)} y_e, \quad \forall j \in V_i \setminus \{i\}, \tag{9}$$

$$x_j, y_e \in \{0,1\}, \quad \forall j \in V_i, \forall e \in E_i. \tag{10}$$

Here b is the *remaining* knapsack capacity for this subproblem. Let $z_i^*(b)$ be the optimal objective value to BP$_i(b)$. Clearly, $z_1^*(B)$ gives the optimal objective value to the original BP, and for $i = n$ we have $z_n^*(b) = p_n$ if $b \geq w_n$, and $z_n^*(b) = 0$ otherwise. Then, from the *principle of optimality* [4, 13, 14], we have the following recurrence relation.

$$z_i^*(b) = \max_{e \in E^+(v_i)} \max\{z_{\partial^+ e}^*(b), p_i + z_{\partial^+ e}^*(b - w_i)\}, \tag{11}$$

Let this be maximized at

$$e_i^*(b) = \arg \max_{e \in E^+(v_i)} \{(11)\}, \tag{12}$$

Then, the optimal decision is given by

$$y_e^*(b) = \begin{cases} 1, & \text{if } e = e_i^*(b), \\ 0, & \text{otherwise}, \end{cases} \tag{13}$$

and

$$x_i^*(b) = \begin{cases} 1, & \text{if } p_i + z_{\partial^+ e_i^*(b)}^*(b - w_i) > z_{\partial^+ e_i^*(b)}^*(b) \\ 0, & \text{otherwise}. \end{cases} \tag{14}$$

Then, computing (11) - (14) backward for $i = n-1, n-2, \cdots, 1$, BP is solved in $O(mB)$ time and $O(nB)$ space. We call this **Algorithm** DP.

3 Shift-and-merge method

In Section 2, we observe that $z_i^*(b)$ is a *step function* of b, and such a function is completely characterized by a set of *discontinuity points*. Then, it is natural to consider computing $z_i^*(b)$ only at these points. Indeed, Nemhauser *et al* [11]. developed a *DP algorithm with lists* to solve a binary KP. The same method was also referred to as a *dominance technique* [6]. Here we call it a *shift-and-merge* technique, since it consists of repeated applications of shift and merge operations, and extend it to a DP algorithm to solve BP. Although the worst case complexity is not improved by such an approach, in most cases the actual amount of computation is significantly reduced.

For an arbitrary integer valued step function $z(\cdot)$ defined on the integer interval $[0,B]$, (b,p) is a discontinuity point if $p = z(b)$ and $z(b-1) < p$. Specifically, we assume $(0,0)$ is always a discontinuity point. Let $L(z) = \{(b^1,p^1),(b^2,p^2),\cdots,(b^r,p^r)\}$ be the set of discontinuity points of $z(\cdot)$ satisfying $0 = b^1 < b^2 < \cdots < b^r \le B$. Then, step function $z(\cdot)$ is completely described by $L(z)$, which we refer to as the *list representation* of $z(\cdot)$ [11].

In this section, we translate the recurrence relation (11) in the language of lists. Let $E^+(i)$ be explicitly denoted as $E^+(i) = \{e_1^i, e_2^i, \cdots, e_t^i\}$ with $t = |E^+(i)|$, and for each k ($0 \le k \le t$) we introduce a subproblem of $\mathrm{BP}_i(b)$ as follows.

BP$_{i,k}(b)$:

$$\begin{array}{ll} \text{Maximize} & (6) \\ \text{subject to} & (7)-(10), \text{ and } y_{e_{k+1}^i} = y_{e_{k+2}^i} = \cdots = y_{e_t^i} = 0. \end{array}$$

This is obtained from graph G_i by removing the last $t-k$ arcs of $E^+(i)$, i.e., by considering graph $G_{i,k} = (V_i, E_{i,k})$, where $E_{i,k} = E_i \setminus \{e_{k+1}^i, \cdots, e_t^i\}$. Let $z_{i,k}^*(b)$ be the optimal objective value to $\mathrm{BP}_{i,k}(b)$.

Clearly,

$$z_{i,0}^*(b) \equiv 0, \tag{15}$$

and

$$z_i^*(b) = z_{i,|E^+(i)|}^*(b). \tag{16}$$

Also, from the principle of optimality, we obtain the following relation.

$$z_{i,k}^*(b) = \max\{z_{i,k-1}^*(b), z_j^*(b), p_i + z_j^*(b-w_i)\}, \tag{17}$$

where $j = \partial^+ e^k$.

This means that the optimal objective value to $\mathrm{BP}_{i,k}(b)$ is given as the maximum value obtained by the following three alternatives.

A_1. Do not take e_k^i. In this case we take an arc in $\{e_1^i, \cdots, e_{k-1}^i\}$ with the corresponding objective value $z_{i,k-1}^*(b)$.

A_2. Take e_k^i and go to node v_j without accepting item i.

A_3. Accept item i, and take e_k^i to go to node v_j.

Let L^0, L^1, L^2 and L^x be the list representations of $z^*_{i,k-1}(b), z^*_j(b), p_i + z^*_j(b - w_i)$, and $z^*_{i,k}(b)$ respectively. We note that L^2 is obtained by shifting L^1 by (w_i, p_i). That is, if $L^1 = \{(0,0),(b^1,p^1),\cdots,(b^r,p^r)\}$, we have $L^2 = \{(0,0)\} \cup \{(w_i + b^l, p_i + p^l) \mid w_i + b^l \le B, l = 0,1,\cdots,r\}$. We call this *shift* operation, and denote $L^2 = \text{Shift}(L^1,(w_i,p_i))$.

Next, corresponding to the recurrence relation (17), L^x can be obtained by *vertically merging* L^0, L^1 and L^2. This means taking the *upper envelope* of these three step functions (see Figure 1), and we write this as

$$L^x = \text{Vertical_Merge}(L^0, L^1, L^2).$$

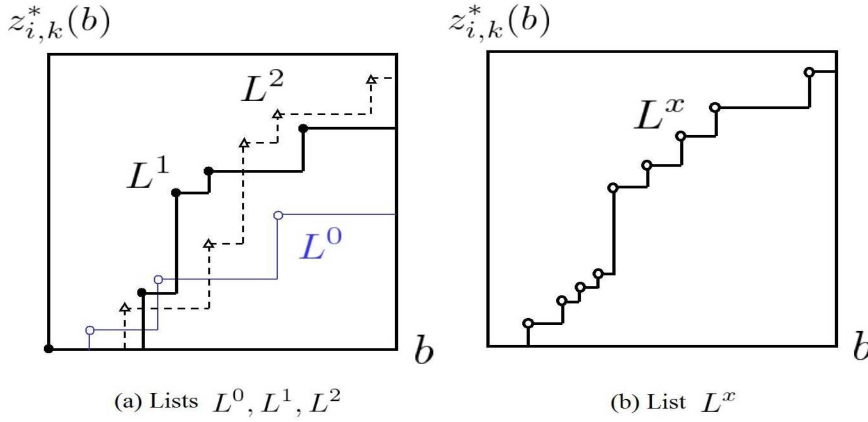

Figure 1: Vertical merge of lists as $L^x = \text{Vertical_Merge}(L^0, L^1, L^2)$.

To compute L^x, we first merge L^0, L^1 and L^2 as $L^x = \{(b^l, p^l) \mid l = 1,2,\cdots,r\}$ in non-decreasing order of b^ls. Then, from L^x we remove all the *dominated* elements. Here we call (b^t, p^t) dominated if there exists some $(b^s, p^s) \in L^x$, satisfying $b^s \le b^t$ and $p^s \ge p^t$.

Then, we present the following *shift-and-merge* DP algorithm. **Algorithm** DP_SM

Input: DAG G and knapsack data $B, (w_j, p_j), j = 1,2,\cdots,n$.
Output: Lists $L_i = L(z^*_i(b))$ for $i = 1,2,\cdots,n$.
Step 1. Set $L_n := \{(0,0),(w_n,p_n)\}$, and $i := n-1$.
Step 2. If $i < 1$, stop.
Step 3. Let $L^0 := \{(0,0)\}$, and $E^+(i)$ be explicitly $\{e^i_1, e^i_2, \cdots, e^i_t\}$, where $t := |E^+(i)|$.
Step 4. For $k = 1,\cdots,t$ do

(i) $j := \partial^+ e^i_k, L^1 := L_j$.
(ii) $L^2 := \text{Shift}(L^1)$.
(iii) $L^x := \text{Vertical_Merge}(L^0, L^1, L^2)$.
(iv) $L^0 := L^x$.

Step 5. Let $L_i := L^0, i := i-1$, and go to Step 2.

Remark 3 DP_SM computes the optimal objective value $z^*_i(b)$, but produces neither optimal $x^*_i(b)$ nor $y^*_e(b)$. To obtain these, we make elements of each list be of the form

(b, p, x^*, e^*), where (x^*, e^*) is the optimal solution to $\mathrm{BP}_i(b)$ at discontinuity point (b, p). Then, in Vertical_Merge(L^0, L^1, L^2) in Step 4 (iii) of the above algorithm, (x^*, e^*) is *inherited* from L_0, in case of A_1, and this is given as $(0, e^i_k)$ and $(1, e^i_k)$, corresponding to the cases of A_2 and A_3, respectively. □

4 Numerical experiments

We implemented DP and DP_SM algorithms in ANSI-C language and conducted computation on a Dell Precision T7400 workstation (CPU: Xeon X5482 Quad-Core×2, 3.20GHz) for various type and size of instances. We also compare the performance of these algorithms against the direct solution by CPLEX [5].

4.1 Design of experiments

We prepare graph $G = (V, E)$ as follows. For each pair (i, j) satisfying $1 \leq i < j \leq n$, we generate an arc (v_i, v_j) randomly with probability $d/(n-1)$. Here d is an integer parameter that controls the number of arcs in G. Since we have $n(n-1)/2$ pairs of nodes, at this stage we have about $m \approx nd/2$ arcs, and the average degree at each node is approximately d. Thus, we call d the *degree parameter*.

Next, for each maximal node $v_i \neq v_1$, we pick up node v_j $(j < i)$ at random and add arc (v_1, v_i) to E. Thus, no maximal nodes remain in G other than v_1. Similarly, we make all nodes other than v_n non-minimal.

We determine the data for items according to the following scheme. The weight w_j is distributed uniformly random over the integer interval $[1, 1000]$, and profit p_j is related to w_j in the following ways.

- Uncorrelated case (UNCOR): uniformly random over $[1, 1000]$, independent of w_j.
- Weakly correlated case (WEAK): uniformly random over $[w_j, w_j + 200]$.
- Strongly correlated case (STRONG): $p_j := w_j + 20$.

4.2 DP results

Table 1 gives the result of computation of DP algorithm with parameters fixed at $B = 5000$ and $d = 3$. The table shows the number of arcs (m), the optimal value (z^*) and the CPU time in seconds (CPU). Each row is the average over 10 randomly generated instances.

The algorithm is able to solve instances with $n \leq 30000$ within a few seconds, irrespective to correlation types. However, it encounters difficulty for the case of $n \geq 40000$, due to the heavy memory requirement of the DP method.

Table 1: DP results ($d = 3, B = 5000$) as average over 10 instances.

n	m	UNCOR		WEAK		STRONG	
		z^*	CPU	z^*	CPU	z^*	CPU
10000	21333	13640.5	0.93	7721.4	0.96	9335.6	1.02
20000	42586	15024.8	1.84	7839.1	1.91	9518.3	2.02
30000	63827	14763.1	2.75	7830.2	2.86	9656.8	3.05

4.3 MIP results

Table 2 summarizes the computation using MIP solver CPLEX 11.1. From the table we see that this solver is able to solve larger instances than the DP algorithm, with the expense of longer CPU time. We set the time limit at 1800 seconds, and for instances with $n \geq 80000$, the solver frequently fails to produce optimal solutions within this time limit.

Table 2: CPLEX results ($d = 3, B = 5000$) as average over 10 instances.

n	m	UNCOR		WEAK		STRONG	
		z^*	CPU	z^*	CPU	z^*	CPU
10000	21333	13640.5	4.25	7721.4	5.35	9335.6	6.11
20000	42586	15024.8	17.42	7839.1	17.88	9518.3	19.84
40000	84968	14998.8	95.90	7873.8	89.39	9777.9	91.80
80000	170412	15814.9	383.34	8094.9	417.16	9978.1	401.50

4.4 DP_SM results

Table 3 gives the result of computation of DP_SM for larger instances with $n \leq 320000$. Here, the column of 'DC%' shows the total number of discontinuity points as percentage over nB. We are able to solve larger instances within much smaller CPU time. Here we observe that the number of discontinuity points increases with the degree of correlation. In strongly correlated case, we usually have more disconrinuity points than in uncorrelated cases, and thus DP_SM is more time consuming than in UNCOR cases.

Table 3: DP_SM results ($d = 3, B = 5000$) as average over 10 instances.

n	m	UNCOR			STRONG		
		z^*	DC%	CPU	z^*	DC%	CPU
10000	21333	13640.5	0.97	0.04	9335.6	10.01	0.46
20000	42586	15024.8	0.93	0.10	9518.3	9.75	0.91
40000	84968	14998.8	0.97	0.22	9777.9	9.76	1.81
80000	170412	15814.9	1.01	0.47	9978.1	12.62	4.71
160000	341313	16317.6	1.13	1.07	10209.3	13.32	10.02
320000	682074	17512.4	1.18	2.26	10464.7	15.41	23.07

5 Conclusion

We formulated the backpacker problem as an extension of the binary knapsack problem, and gave DP and DP_SM algorithms to solve this problem to optimality. These algorithms were implemented in ANSI-C language, and numerical experiments were carried out to evaluate the performance of the developed algorithms. We were able to solve the backpacker problem with up to 320000 items of various correlation types within a few seconds in an ordinary computing environment. Computation was not much influ-

enced by the change of the correlation types between weight and profit of items, and over-performed computation by MIP solver.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[2] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.

[3] G. Cho, D.X. Shaw, A depth-first dynamic programming algorithm for the tree knapsack problem, INFORMS Journal on Computing 9 (1997) 431-438.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, MA, 2001.

[5] CPLEX 11.1, ILOG, http://www.ilog.com/ products/cplex, 2009.

[6] D. El Baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem, Journal of Parallel and Distributed Computing 65 (2005) 74-84.

[7] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman and Company, San Francisco, CA, 1979.

[8] D.S. Johnson, K.A. Niemi, On knapsacks, partitions, and a new dynamic programming technique for trees, Mathematics of Operations Research 8 (1983) 1-14.

[9] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, Berlin, 2004.

[10] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley and Sons, New York, NY, 1990.

[11] G.L. Nemhauser, Z. Ullmann, Discrete dynamic programming and capital allocation, Management Science 15 (1969) 494-505.

[12] N. Samphaiboon, T. Yamada, Heuristic and exact algorithms for the precedence-constrained knapsack problem, Journal of Optimization Theory and Application 105 (2002) 659-676.

[13] R. Sedgewick, Algorithms in C, 3rd Edition, Addison-Wesley, Reading, 1998.

[14] L.A. Wolsey, Integer Programming, John Wiley & Sons, New York, NY, 1998.

[15] B.-J. You, T. Yamada, A pegging approach to the precedence-constrained knapsack problem, European Journal of Operational Research 183 (2007) 618-632.