Research Article

International Journal of Advanced Computer Research, Vol 7(28)
ISSN (Print): 2249-7277 ISSN (Online): 2277-7970
http://dx.doi.org/10.19101/IJACR.2017.728003

Finding the most efficient paths between two vertices in a knapsack-item weighted graph

Nadav Voloch*
Department of Computer Science, The Open University of Israel, Ra'anana, Israel

Received: 25-September-2016; Revised: 22-November-2016; Accepted: 28-November-2016

Abstract
There have been several combinations of the knapsack problem and the shortest paths on weighted graph problems in different researches. The combination is often used to describe the choices made during the knapsack problem stages using dynamic programming methods, by using the knapsack graph. But these researches consider only two aspects of weight and value for an item/vertex. The objective of this paper is to address a different kind of problem in which we are taking into consideration three properties: item weight, item value and edge weight (that connects two items, but its weight is not depended on its vertices). The problem presented here is finding the most efficient path between two vertices of this specific kind of graph, in three aspects- minimal edge wise, maximum knapsack value wise, and a combination of maximal efficiency of both properties. This is done through an object oriented method, in which every path of the graph, between two chosen vertices, has comparable attributes, that gives us the ability to prefer a certain path from another. An algorithm for finding these optimal paths is presented here, along with specific explanations on its decision stages, and several examples for it. The results were achieved an exact paradigm for the integrated problem, taking into consideration any desired aspect, and achieving optimal choices per each attribute.

Keywords
Knapsack problem, Shortest paths on weighted graphs, Dijkstra's algorithm, 0-1 knapsack problem, Graph theory, Dynamic programming, All paths between two vertices in a graph.

1.Introduction
The research is within the scope of theoretical algorithms. In this branch, there are two well-known problems that are:

A. The knapsack problem dating back far as more than a century ago, in which, for a set of items, we have to determine how many items of every type include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible as reviewed in [1]. A knapsack item is described as follows:

- An item, that represent an object of some sort (usually, merchandise).
- The item has at least two attributes (could be more) of weight (w_i) and value (v_i).
- The item's attributes are comparable to other items' attributes.

For this problem there are three main versions:
1. The most common problem being solved is the 0-1 knapsack problem, which restricts the number of copies of each kind of item for 0 or 1 (hence its name), meaning we are able to pick only one item from each kind. This is the specific problem to which this paper addresses.
2. The bounded knapsack problem (BKP) in which there is no limit of one copy per item, but a limit for the number of copies of each kind of item extends to a maximum finite amount.
3. The unbounded knapsack problem (UKP) places no upper bound on the number of copies of each kind of item.

B. The shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized as summarized in [2].

There are several algorithms that handle these problems, some more efficient than others.

*Author for correspondence

For example, to UKP there is a greedy approximation algorithm proposed by George Dantzig (as described in [3]). In the Dantzig algorithm, we take the value of an item (v_i), and its weight (w_i), and create an attribute of efficiency that is the proportion of v_i / w_i, we arrange them in a decreasing order, then greedily pick the item that have the best efficiency rate and put as much as we can from it to the knapsack, until we cannot fit anymore to the knapsack, and then we proceed to the next item on the efficiency list. This greedy algorithm guarantees at least a result of $m/2$ (m is the knapsack limit).

The 0-1 knapsack problem is usually solved by a pseudo-polynomial dynamic programming algorithm that can be seen in [4]. In this algorithm, at any stage, we take the items that fit the current knapsack state's best, thus creating a close to optimal solution of the problem.

A solution for the shortest path problem is the well-known Dijkstra's algorithm, which finds the shortest path between nodes in a graph, conceived by Edsger W. Dijkstra (as presented in [5]). The algorithm was optimized in many researches such as [6].

There have also been papers about dynamic programming solutions for the knapsack problem, using shortest path problem, with the creation of a knapsack graph such as [7] and [8], but these papers gave a solution to the knapsack problem using the shortest path problem. A problem arises when a knapsack graph, that its edge weights are non-dependent in the vertex values, is built.

The decision of finding the most efficient path, attributing the knapsack-items as vertices, has to take additional factors into consideration – the knapsack weight limit, and maximizing the value of items. This situation is viable for real-life circumstances, in which a path has a non-dependent attribute (physical distance, travel time, etc.), and there are different kinds of items to be picked in different locations on this path.

For example, a sales person that wishes to travel from a certain point to another, with several possible places on his way in which he can pick up different kinds of supply (items), would like to optimize his path, given his weight bound for merchandise.

While most papers mentioned above, consider only two aspects of knapsack weight and value, here we take into consideration three properties: item weight, item value and edge weight (that connects two items, but its weight is not depended on its vertices). We find the different possibilities of paths from a source vertex to a target vertex, considering the preferred attribute we choose to apply.

The objective of this research is to analyze and define this specific problem, and to give an optimal solution for it, considering all of the elements mentioned, and in the constraints that apply these two integrated problems.

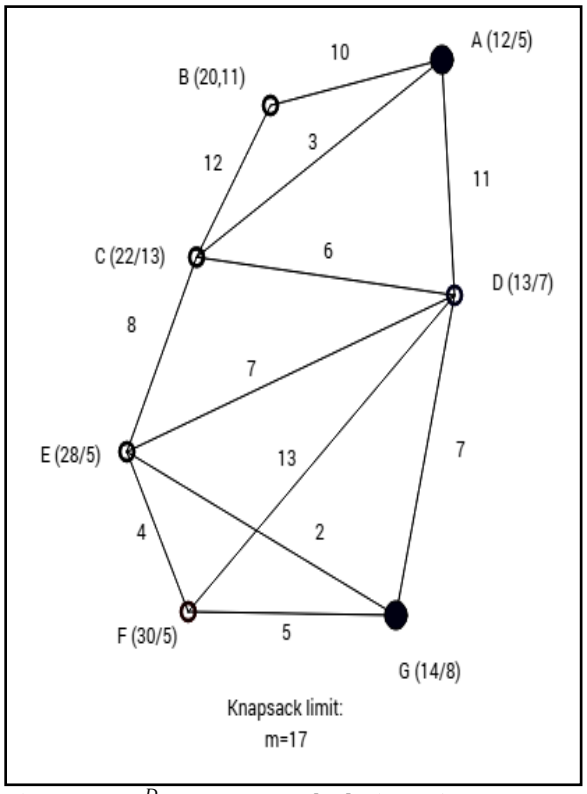

Figure 1 G^D-Vertices marked (v_i / w_i), source-A, target-G

2.The knapsack weight-independent graph

Given a weighted graph $G = (V, E)$, where V represents the vertices of the graph, and E its edges, we first constrain the edges' weights to be non-negative ones. The second stage is adding properties of weight (w_i) and value (v_i) to each vertex, turning it to a legitimate knapsack item. We take into consideration the difference between the weight of the vertex, that is marked w, as mentioned before, and the weight of the edge, that will be marked as w^E. In addition to these properties, we will add to the graph G a limit m, that represents the knapsack limit

of maximum weight. We now have a suitable graph for both problems (knapsack and shortest paths in weighted graphs), that will be marked G^D.

3.Finding the most efficient path between two vertices

Given a graph G^D as described above, we wish to find a path between two chosen vertices (source vertex-v_s and target vertex-v_t), that with a given knapsack weight limit (m), will achieve optimal results, by choosing the maximal value of items from this path. An example of such a G^D is shown in *Figure 1*, where vertices are presented as $V(v_i/ w_i,)$, the source vertex is A, and the target vertex is G. Here we divide the meaning of "optimal" into three different cases:

a. The main priority is given to the minimal edge weight between the two vertices. In this specific case, a possible solution is a two-step algorithm, in which we find the shortest path between two vertices in G^D, by using Dijkstra's algorithm, and then we take the vertices from the chosen sub-graph (the path), limiting the vertex weights with m- the knapsack limit, and then use the pseudo-polynomial dynamic programming algorithm for the 0-1 knapsack, to choose the vertex-items from the path.
b. The main priority is given to the maximal knapsack value of items in the path between the two vertices. In this case, we have to take into consideration all of the paths between the two vertices, to achieve an optimal knapsack-value choice of item-vertices.
c. An equal priority is given to both aspects minimal edge weight, and maximal knapsack value of items. In this case we also have to consider all possible paths between two vertices, and calculate the optimal difference between the two aspects.

4.The algorithm for the combined problem

For solving the problem described above, in all of its three different cases, we use an algorithm that first finds all of the possible paths between the two vertices, and then gives attributes to each path, that will help us choose the optimal path. These attributes are: total edge weight of the path, and total value of knapsack chosen items, given a knapsack limit (m). For the first part, of finding all possible paths, we can use the Ford-Fulkerson algorithm (as seen in [9]), or another one that finds all of the paths from a source vertex to a target vertex. For the second part we use the pseudo-polynomial dynamic programming algorithm mentioned above, to choose the vertex-items from every path. The last step is choosing the priority case (*a*, *b* or *c*, as described in the previous part).

The algorithm is as follows:
Finding most efficient path in a knapsack-item weight-independent graph (Graph GD, Vertex source, Vertex target, integer max_item_weight, char priority):

- Create a list of all paths source to target L^{path} = Ford-Fulkerson (G^D, source, target)
- For each path L_i^{path} set attribute of total edge weight $tew_i = \sum w^E{}_i (L_i^{path})$
- For each path L_i^{path} create chosen vertex-items list $L^K(L_i^{path})$ = Knapsack 0-1 (V (L_i^{path}), max_item_weight)
- For each $L^K(L_i^{path})$, set attributes of total knapsack value $tkv_i = \sum v_i (L^K(L_i^{path}))$
- If (priority='a')
  - Find min tew_i (L^{path})
  - Return L_i^{path}
- If (priority='b')
  - Find max tkv_i (L^{path})
  - Return L_i^{path}
- If (priority='c')
  - Find max (tkv_i - tew_i)(L^{path})
  - Return L_i^{path}

As seen in the algorithm, in the first stage we have to create a list of all of the paths, this is done by the Ford-Fulkerson algorithm that gives us all of the paths between two desired vertices. The paths are the main objects of this procedure, because we choose one that is most fit to our priority. At the second stage we set the paths' unique property of total edge weight (tew_i) by summing up all of the edge weights. At the third stage we use the procedure of the pseudo-polynomial dynamic programming algorithm of knapsack-item picking. This is done for the purpose of the fourth stage, and allows us to see what chosen items we take, to optimize the result of the procedure. In the fourth stage we set the unique path property of total knapsack value (tkv_i) by summing up all of the paths vertex-items values.

These properties are the ones that will help us choose the optimal path per a given priority. The fifth and last stage is choosing the desired path by the given priority: in case the priority is *'a'*, we will choose the path that has the minimal tew_i, thus getting the shortest path between the two vertices. In case the

priority is *'b'*, we will choose the path that has the maximal tkv_i, thus getting the path that gives us the optimal value for items between the two vertices. In case the priority is *'c'*, we will choose the path that has the best integrated value of tkv_i - tew_i, thus getting the path that gives us the optimal property-integrated value between the two vertices. The working procedure of the algorithm is portrayed in a flowchart in *Figure 2*.

Table 1 G^D scheme example no.1

G^D- knapsack-item weight-independent graph		
Vertices		
V(G^D)	***v***	***w***
A	12	5
B	20	11
C	22	13
D	13	7
E	28	5
F	30	5
G^D- knapsack-item weight-independent graph		
Vertices		
G	14	8
Edges		
E(G^D)	***w^E***	
A-B	10	
B-C	12	
C-D	6	
A-C	3	
C-E	8	
D-E	7	
E-F	4	
F-G	5	
E-G	2	
D-F	13	
D-G	7	
A-D	11	
m	**Source vertex**	**Target vertex**
17	*A*	*G*

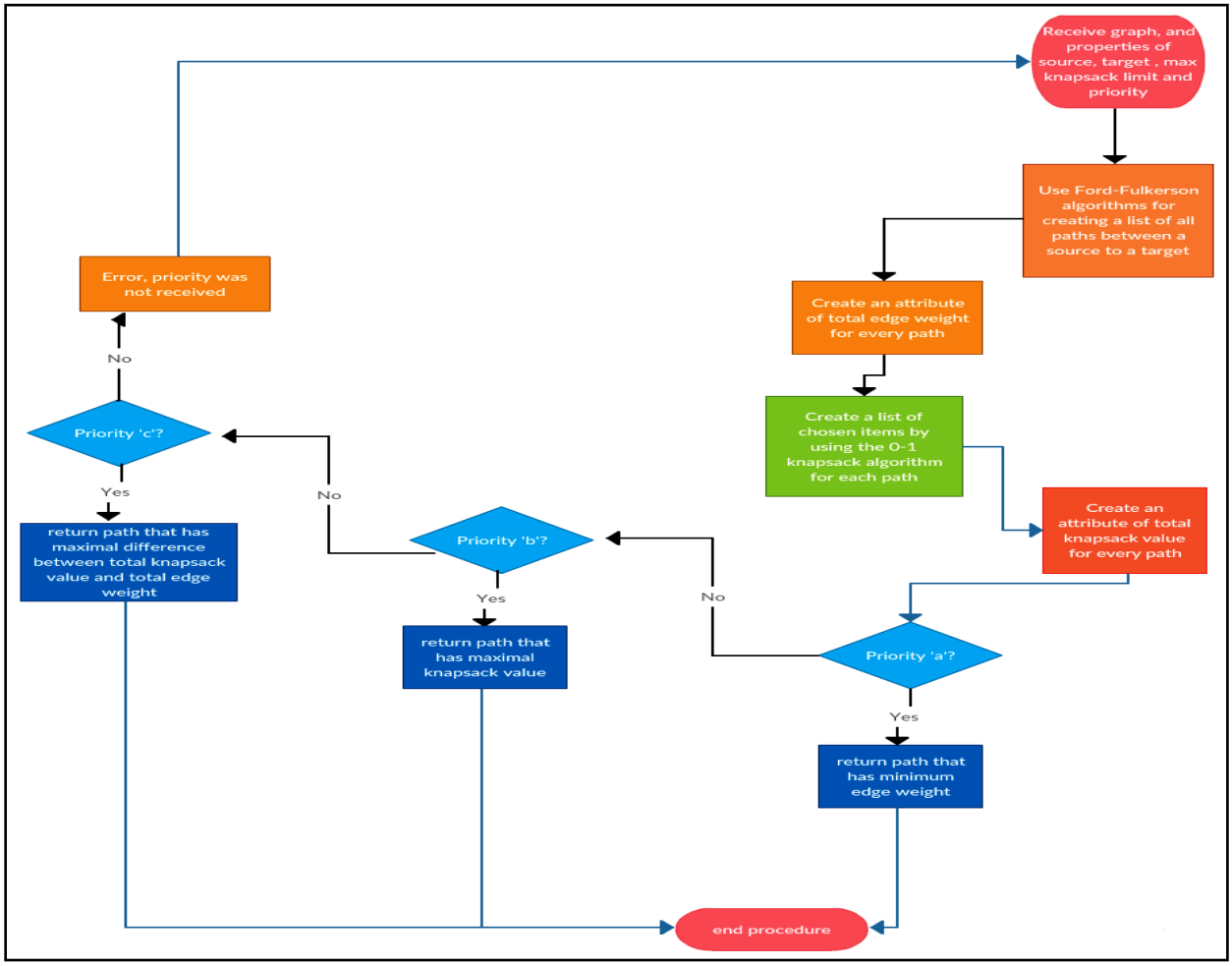

Figure 2 Finding most efficient path in a knapsack-item weight-independent graph-the algorithm

For example, given the graph G^D, shown in *Figure 1*, and its data are presented in *Table 1*, using *A* as the source vertex and *G* as the target vertex, and a knapsack weight limit of *m*=17, the first step is finding all of the paths between *A* and *G*.

There are 16 possible paths between *A* and *G* in G^D, L^{path} is presented in *Table 2*, with all of the properties of the paths, including the chosen knapsack items, tkv_i, and tew_i as described above. In every path we choose the vertices that maximize their knapsack value these are the chosen items on the path that add up to tkv_i, while the weights of the edges is summed up to tew_i. In addition, there are the three paths chosen by the different priorities *('a', b'*, and *'c'* as described above). The three chosen paths are seen in *Figure 3* ($L_i^{path}(G^D)$, priority *'a'* (minimal edge weight)), *Figure 4* ($L_i^{path}(G^D)$, priority *'b'* (maximal knapsack value)), and *Figure 5* ($L_i^{path}(G^D)$, priority *'c'* (maximal difference between knapsack value and edge weight)). Looking at the table and figures, we can now see that for priority *'a'*, the minimal edge weight achieved by L_{12}^{path} is 13, and that for priority *'b'*, the maximal knapsack value achieved by L_{13}^{path} is 71, and that for priority *'c'*, the maximal difference between the knapsack value and edge weight achieved by L_{11}^{path} is 70-20=50. The shaded vertices in the figured are the chosen items.

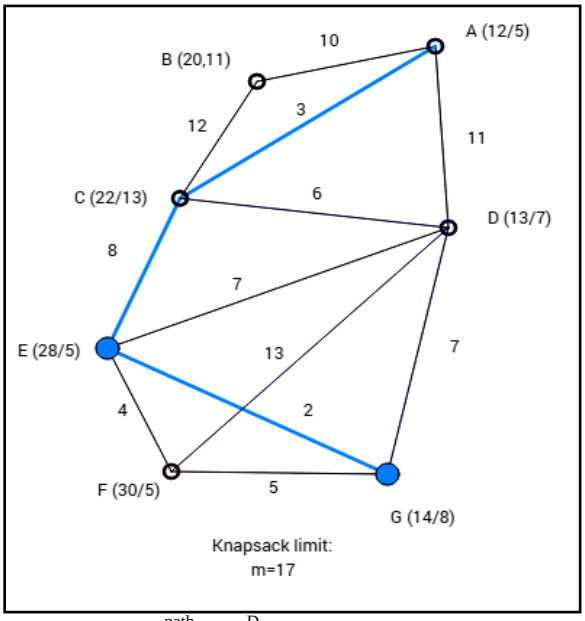

Figure 3 L_{12}^{path} (G^D), priority 'a' (minimal edge weight)

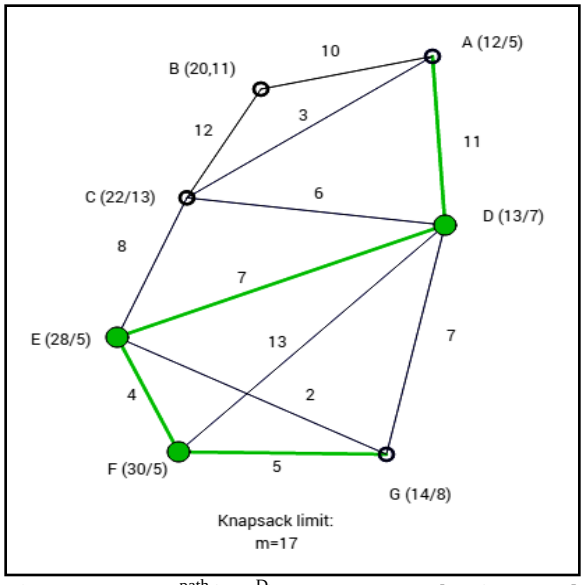

Figure 4 L_{13}^{path}(G^D) , priority 'b' (maximal knapsack value)

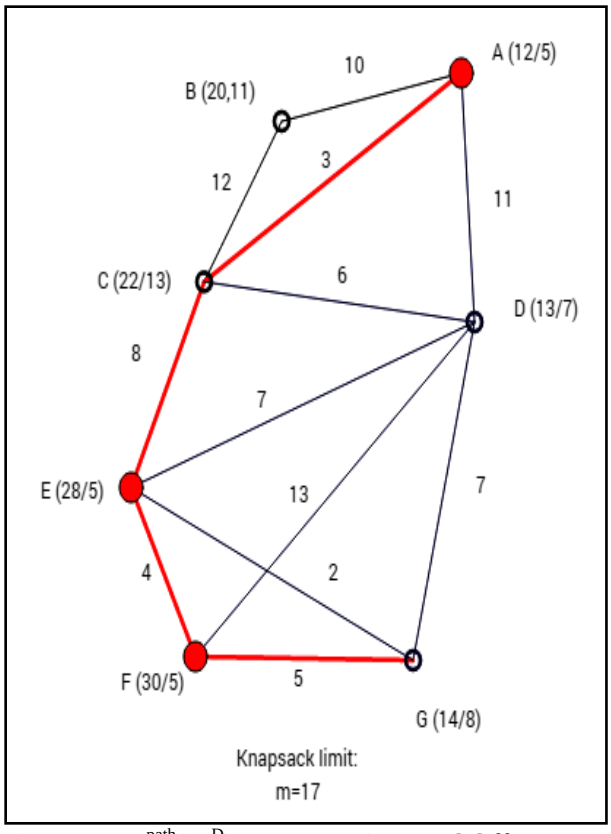

Figure 5 L_{11}^{path}(G^D), priority 'c' (maximal difference between knapsack value and edge weight)

Table 2 L^{path} for G^D scheme example no.1

L^{path} (G^D) For A→G, with m=17				
Paths				
No.	**V(L^{path})**	**Chosen items**	**tkv_i**	**tew_i**
1	*A→B→C→D→E→F→G*	*D, E, F*	71	44
2	*A→B→C→D→E →G*	*A, D, E*	53	37
3	*A→B→C→D→F→G*	*A, D, F*	55	46
4	*A→B→C→D→ G*	*A, B*	32	35
5	*A→B→C→EF→ G*	*A, E, F*	70	39
6	*A→B→C→E→ G*	*B, E*	48	32
7	*A→C→D→E→F→ G*	*D, E, F*	71	25
8	*A→C→D→E →G*	*A, D, E*	53	18
9	*A→C→D→F →G*	*A, D, F*	55	27
10	*A→C→D→G*	*D, G*	27	16
11	*A→C→E→F →G*	*A, E, F*	70	20
12	*A→C→E →G*	*E, G*	42	13
13	*A→D→E→F →G*	*D, E, F*	71	27
14	*A→D→E →G*	*A, D, E*	53	20
15	*A→D →F →G*	*A, D, F*	55	29
16	*A→D →G*	*D, G*	27	18
Chosen L_i^{path} by priority				
Priority	**L_i^{path}**			
a	**No.12**			
b	**No.13**			
c	**No.11**			

5.More results and examples for G^D

In *Table 3*, we can see a another example for GD – its item-vertices by their properties of value (v) and weight (w) and the edges by their weight (wE), then the source and the target vertices are shown, and the knapsack weight limit (m).

Table 4 is ordered in the same manner as *Table 2* described above, and displays a different, smaller graph GD and its result after performing the algorithm, and choosing the paths by the three different priorities.

6.Discussion

The results of this study are different from the ones mentioned above as far as knapsack graphs are being dealt with. In a regular knapsack graph the edge weight is solely dependent on the item vertices themselves, thus being built dynamically along with the choices of items for the knapsack problem. Here we see that a different situation is being handled, in which the edges are known in advance, and so are the item-vertices, thus allowing us to achieve optimal results considering all possible path options for every item-combination possible.

In addition, a specific case of traveling between two vertices of a graph is being described, and we can see that there are different results for different kinds of priority, and we can adjust our desired result for different cases of attribute preferences. The analysis of the results shows us that on the same G^D, totally different paths are being chosen, and the optimal results in every case are different, as shown in the figures and tables.

Table 3 G^D scheme example no.2

G^D- knapsack-item weight-independent graph		
Vertices		
$V(G^D)$	***v***	***w***
A	32	13
B	40	9
C	42	10
D	33	27
E	38	8
F	37	9
G	17	9
Edges		
$E(G^D)$	**w^E**	
A-B	16	
B-C	12	
C-D	6	
A-C	8	
C-E	18	
D-E	7	
E-F	14	
F-G	5	
E-G	20	
D-F	12	
D-G	4	
A-D	3	
m	**Source vertex**	**Target vertex**
30	*A*	*G*

7.Scope and optimization of the algorithm

In cases of equality between two possible paths, for all three possible priority types, we find ourselves in a bit of a problem. For example, in *Table 2*, we can easily see that for priority *'b'* (maximal knapsack value), there are two possible solutions L_1^{path} and L_{13}^{path}, both achieve tkv_i of 71. In a case like this, the algorithm has a supplement of checking the other property- meaning tew_i for this case of *'b'* priority, thus preferring L_{13}^{path}, achieving tew_i of 27 over L_{13}^{path}, achieving tew_i of 44. For priority *'a'*, in case of equality, the tkv_i is checked to prioritize, and for priority *'c'*, any of the properties of tew_i and tkv_i can be checked.

8.Limitations of the study

The algorithm presented for G^D is viable in cases of a well-known situation, in which all of the items in the graph have the needed attributes, and they are known to the traveler (that goes from the source vertex to the target one). In cases that have an unknown attribute or attributes in the items, the problem is much more complex, and not in the scope of this study. Another case that is not covered is the case in which the vertices are sets of items, with different attributes of weight and value. In that case, the algorithm might be well served as a base for the solution, but needs a major adaptation to be suitable for this problem.

Table 4 L^{path} for G^D scheme example no.2

L^{path} (G^D) For A→G, with m=17				
Paths				
No.	**V(L^{path})**	**Chosen items**	**tkv_i**	**tew_i**
1	*A→B→C→D→E→F→G*	*B, C, E*	120	60
2	*A→B→C→D→E →G*	*B, C, E*	120	61
3	*A→B→C→D→F→G*	*B, C, F*	119	51
4	*A→B→C→D→ G*	*B, C, G*	99	38
5	*A→B→C→EF→ G*	*B, C, E*	120	65
6	*A→B→C→E→ G*	*B, C, E*	120	66
7	*A→C→D→E→F→ G*	*C, E, F*	117	40
8	*A→C→D→E →G*	*C, E, G*	97	41
9	*A→C→D→F →G*	*C, F, G*	96	31
10	*A→C→D→G*	*A, C*	74	18
11	*A→C→E→F →G*	*C, E, F*	117	45
12	*A→C→E →G*	*C, E, G*	97	46
13	*A→D→E→F →G*	*A, E, F*	107	29
14	*A→D→E →G*	*A, E, G*	87	30
15	*A→D →F →G*	*A, F*	69	20
16	*A→D →G*	*A, G*	49	7

Chosen L_i^{path} by priority	
Priority	**L_i^{path}**
a	**No.16**
b	**No.1**
c	**No.13**

9.Conclusion and future work

For the graph presented here G^D, we have seen that the most efficient path differs in case of different types of priority, and the results given by different priorities show us that the shortest path is not necessarily the most efficient one. For this algorithm, there could be many more applications and expansions, like using the vertex labels as strings, or specific data structures (a stack on each vertex, for example). A system, written in Java, was built for manufacturing the results shown in this paper, based on the algorithm described in the previous parts. Improving and expanding it as described here is a work in progress.

Acknowledgment

None.

Conflicts of interest

The author has no conflicts of interest to declare.

References

[1] Poirriez V, Yanev N, Andonov R. A hybrid algorithm for the unbounded knapsack problem. Discrete Optimization. 2009;6(1):110-24.

[2] Abraham I, Fiat A, Goldberg AV, Werneck RF. Highway dimension, shortest paths, and provably efficient algorithms. In proceedings of the ACM-SIAM symposium on discrete algorithms 2010 (pp. 782-93) Society for Industrial and Applied Mathematics.

[3] Ensthaler L, Giebe T. Subsidies, Knapsack auctions and dantzig's greedy heuristic. 2009.

[4] Pisinger D, Sigurd M. Using decomposition techniques and constraint programming for solving the

two-dimensional bin-packing problem. INFORMS Journal on Computing. 2007;19(1):36-51.
[5] Yan, M. http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf. Accessed 23 May 2016.
[6] Mehlhorn K, Sanders P. Algorithms and data structures: the basic toolbox. Springer Science & Business Media; 2008.
[7] Druken BK. A hike through the forest: the knapsack problem in graph theory. Senior Honors Projects. 2008.
[8] Pferschy U, Schauer J. The knapsack problem with conflict graphs. Journal of Graph Algorithms and Applications. 2009;13(2):233-49.
[9] Jiang Z, Hu X, Gao S. A parallel ford-fulkerson algorithm for maximum flow problem. In proceedings of the international conference on parallel and distributed processing techniques and applications 2013 (pp. 70-3). WorldComp.

Nadav Voloch is a research student towards an M.Sc in computer science in the Open University in Israel. His B.S degree in computer science is from Sapir Academic College in Israel. He is a lecturer in the Center for Academic studies in Or-Yehuda, Israel, in the information systems and computer science departments, and in Ruppin academic center, Israel in the department of electric and computer engineering, as well as adjunct faculty in the Academic college Tel Aviv – Yaffo, Israel. His research interests include graph algorithms, knapsack type problems, databases and data structures.
Email: nvolloch@yahoo.com