Optimal paths of knapsack-set vertices on a weight-independent graph

NADAV VOLOCH
Department of Computer Science
The Open University
University Road 1, Ra'anana, 4353701
ISRAEL
nadavvo@mta.ac.il http://www.openu.academia.edu/NadavVoloch

Abstract: - Two problems that are often studied and researches in computer science algorithms are the knapsack problem and the shortest paths on weighted graphs problem. Their combination is often addressed by dynamic programming solutions for the knapsack problem, using shortest path problem, with the creation of a knapsack graph. But these researches consider only two aspects of weight and value for an item/vertex, and here we 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). Every vertex in this specific graph is a set of knapsack 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. The problem presented here is finding the most efficient path between two vertices (source and target), in three aspects- minimal edge wise, maximum knapsack value wise, and a combination of maximal efficiency for both properties. An algorithm for finding these optimal paths is presented here along with specific explanations on its decision stages, and several examples for it.
.

Key-Words: - Knapsack problem; Shortest paths on weighted graphs; Dijkstra's algorithm; 0-1 knapsack problem; All paths between two vertices in a graph;

1 Introduction

Within the scope of theoretical algorithms there are two well known problems that are:

a. **The knapsack problem** dating back far as more than a century ago ([1]), which is that for a set of items, each with a weight and a value, we have to determine the number of each item to 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. The name is based on a real-life problem in which someone has weight-limited knapsack, and is supposed to maximize the value of given fixed-size items that are put in the knapsack. It is a problem in combinatorial optimization and it arises in resource allocation aspects that include financial or other allotted criteria restraints. Knapsack-type problems are portrayed in decision-making processes, such as selecting financial portfolios, and cutting of raw materials in the least wasteful manner.
To 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 to 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]. The shortest path problem can be defined for directed or undirected graphs, and even for mixed ones. In this paper we relate to the undirected graphs definition of the problem. Shortest path algorithms are used for several applications such as finding the shortest path between two physical locations (for geographic navigation applications), finding the shortest path of routing in network communication. In civil engineering this problem is often used to manifest the roads as the graph's edges, and the nodes as intersections, the edges' weights can represent to the length of the road or the time needed to travel it. Other applications of the problem are designing facility layouts, robotics and VLSI design.

There are several algorithms that handle these problems, some more efficient than others.
For example, to UKP there is a greedy approximation algorithm proposed by George Dantzig ([3]). His version sorts the items in decreasing order of value per unit of weight, v_i/ w_i, where v represents the value of each item i of n items (for $1 \leq i \leq n$) and w represents its weight . It then proceeds to insert them into the sack, starting with as many copies as possible of the first kind of item until there is no longer space in the sack for more. Provided that there is an unlimited supply of each kind of item, if m is the maximum value of items that fit into the sack, then the greedy algorithm is guaranteed to achieve at least a value of $m/2$.
For the 0-1 knapsack problem there is a pseudo-polynomial dynamic programming algorithm, suggested in different papers such as [4], in which at every step, the maximization process is re-assessing the current status limit-wise, and optimizing the result, by picking the best item for the current stage.
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 ([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], and also with other applications of it such as [9].
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 vertices values, is built. This situation can apply 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.
The decision of finding the most efficient path, attributing sets of knapsack-items as vertices, has to take additional factors into consideration – the knapsack weight limit, and maximizing the value of items. 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). A graph having the three attributes mentioned above was first shown in [10], and in it the vertices themselves were knapsack items (singular). Here we find the different possibilities of paths from a source vertex to a target vertex, having the vertices as sets of knapsack items, considering the preferred attribute we choose to apply.

2 Problem Formulation- 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 to each vertex a set of items with the properties of weight (w_i) and value (v_i) to all of these knapsack items. We take into consideration the difference between the weight of the items in 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 the integrated problem that is defined as finding the most efficient path between two vertices that are sets of knapsack items, which will be marked G^D. An example of such a G^D is shown in Fig.1, were the items in the vertices are presented as w_i/v_i, the source vertex is A, and the target vertex is F.

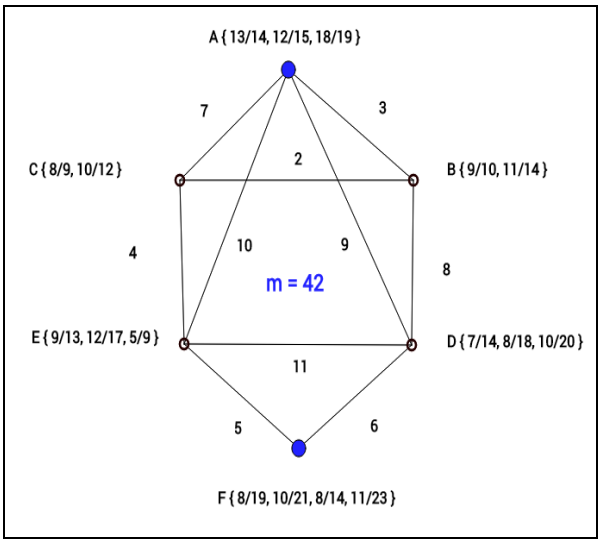

Fig. 1. *G^D – Items in the vertices marked w_i/v_i , source-A, target- F*

2.1 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.

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.**
   For 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 unite the items in the vertices from the chosen sub-graph (the path), limiting the vertices weights with m- the knapsack limit, and then use the pseudo-polynomial dynamic programming algorithm for the 0-1 knapsack, to choose the 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 items from all of the 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.

3 Problem Solution- The algorithm for the integrated problem

For solving the problem described above, on 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 ([11]), or another one that finds all of the paths from a source vertex to a target vertex, such as the Edmonds–Karp algorithm ([12]) for computing the maximum flow in a flow network in $O(V E^2)$ time, V being the number of vertices in the graph/network, and E being the number of edges, or the Dinic's algorithm ([13]), also for a maximum flow network, that achieves a better time of $O(V^2 E)$.
Since the Dinic's algorithm is the most efficient one, we will use it in our algorithm.
For the second part we unite all of the items of the vertices to one set and then use the pseudo-polynomial *Knapsack 0-1* dynamic programming algorithm, to choose the items from every path.
The last step is choosing the priority case (*a, b* or *c*, as describes in the previous part).
During this process, it is convenient to manifest all of the attributes to the complex object of **path**.

The algorithm is as follows:

Finding most efficient path in a knapsack-item weight-independent graph **(Graph G^D, Vertex *source*, Vertex *target*, integer *max_item_weight*, char *priority*):**

1. Create a list of all paths *source* to *target* :

   1.1. $L^{path} = Dinic\ (G^D, source, target)$

2. For each path L_i^{path} where $1 \le i \le |\ L_i^{path}\ |$ set attribute of total edge weight:

   2.1. $tew_i = \sum_{j=0}^{|E|} w^E{}_j\ (L_i^{path})$

3. For each path L_i^{path} create chosen items list:

   3.1. $L^K(L_i^{path}) = Knapsack\ 0\text{-}1\ (\ V\ (L_i^{path}),\ max\_item\_weight\ )$

4. For each $L^K(L_i^{path})$, set attributes of total knapsack value:

   4.1. $tkv_i = \sum_{j=0}^{|L\_Path|} v_j\ (L^K(L_i^{path}))$

5. If (*priority='a'*):

   5.1. Find min $tew_i\ (L^{path})$
   5.2. Return L_i^{path}

6. If (*priority='b'*):

   6.1. Find max $tkv_i\ (L^{path})$
   6.2. Return L_i^{path}

7. If (*priority='c'*):
   7.1. Find max $(tkv_i - tew_i)(L^{path})$
   7.2. Return L_i^{path}

For example, given the graph G^D, shown in Fig.1, and its data as presented in Table 1, using A as the source vertex and F as the target vertex, and a knapsack weight limit of m=42, the first step is finding all of the paths between A and F. An item is presented as its vertex and an index $1 \le i \le |V_i|$, e.g. the items of vertex A are marked A_1, A_2, A_3. An edge is marked by its source vertex and its target vertex, e.g. *A-B*. There are 7 possible paths between A and F 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 vertex- items that maximize the knapsack value- these are the chosen items of 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 Fig.2 ($L_i^{path}\ (G^D)$, priority *'a'* (minimal edge weight)), Fig.3 ($L_i^{path}\ (G^D)$, priority *'b'* (maximal knapsack value)), and Fig.4 ($L_i^{path}\ (G^D)$, 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_3^{path} is 14, and that for priority *'b'*, the maximal knapsack value achieved by L_7^{path} is 90, and that for priority *'c'*, the maximal difference between the knapsack value and edge weight achieved by L_6^{path} is 88-15=73. The bold items in the figures are the chosen items.

3.1 Complexity analysis

We will now analyze the different parts of the algorithm as described in the previous part, considering V being the number of vertices in the graph/network, E being the number of edges, m the maximal knapsack weight, and n the number of knapsack-items.

The first stage of the algorithm is finding all of the paths using the *Dinic's* algorithm. This stage has an $O\ (V^2 E)$ running time as explained in the previous part. The second stage of the algorithm is finding the entire edge weight total of the paths, and its approximation is $O\ (E)$. For the third stage we use the pseudo-polynomial *Knapsack 0-1* dynamic programming algorithm that has an $O\ (nm)$ complexity**,** and similarly to stage 2, in the fourth stage we have to set the entire edge weight total of the paths' knapsack-items, so its approximation is of $O\ (nE)$ complexity. For stages 5-7 (we choose only one stage) we have to find the max attribute of the paths created in the previous stages, hence achieving a complexity of $O\ (E)$**.** The summation of these stages gives us the complexity of $O(V^2E)+ 2O(E)+ O\ (nm)+ O\ (nE)$, meaning $O((m+V^2)E+mn)$.

For example, we can take the graph presented in Fig.1, that its scheme is presented in Table 1, and calculate the estimated running time by the given parameters:

$V = 6$, $E = 10$, $m = 42$, and $n = 17$.

The calculation is as follows:

$(m + V^2)\ E + mn = (42 + 6^2) \cdot 10 + 42 \cdot 17 =$

$780 + 714 = 1494.$

This means that for the graph shown in Fig.1, that its scheme is presented in Table 1, the estimation of the number of elementary operations performed by the algorithm is 1494, where an elementary operation takes a fixed amount of time to perform.

Table 1: G^D Scheme Example No.1

G^D- knapsack-item weight-independent graph		
Vertices-items (V_i)		
$V(G^D)_i$	v	w
A_1	14	13
A_2	15	12
A_3	19	18
B_1	10	9
B_2	14	11
C_1	9	8
C_2	12	10
D_1	14	7
D_2	18	8
D_3	20	10
E_1	13	9
E_2	17	12
E_3	9	5
F_1	19	8
F_2	21	10
F_3	14	8
F_4	23	11
Edges		
$E(G^D)$	w^E	
A-B	3	
B-C	2	
A-C	7	
A-E	10	
B-D	8	
C-E	4	
D-E	11	
E-F	5	
D-F	6	
A-D	9	
m	Source vertex	Target vertex
42	A	F

Table 2: L^{PATH} for G^D Scheme Example No.1

L^{path} (G^D) For A→F, with m=42				
Paths				
No.	$V(L^{path})$	Chosen items	tkv_i	tew_i
1	$A \rightarrow B \rightarrow D \rightarrow F$	D_1, D_2, F_1, F_3, F_4	88	17
2	$A \rightarrow B \rightarrow D \rightarrow E \rightarrow F$	D_2, E_3, F_1, F_2, F_4	90	27
3	$A \rightarrow B \rightarrow C \rightarrow E \rightarrow F$	E_3, F_1, F_2, F_3, F_4	86	14
4	$A \rightarrow C \rightarrow E \rightarrow F$	E_3, F_1, F_2, F_3, F_4	86	16
5	$A \rightarrow E \rightarrow F$	E_3, F_1, F_2, F_3, F_4	86	15
6	$A \rightarrow D \rightarrow F$	D_1, D_2, F_1, F_3, F_4	88	15
7	$A \rightarrow D \rightarrow E \rightarrow F$	D_2, E_3, F_1, F_2, F_4	90	25
Chosen L_i^{path} by priority				
Priority	L_i^{path}			
a	No.3			
b	No.7			
c	No.6			

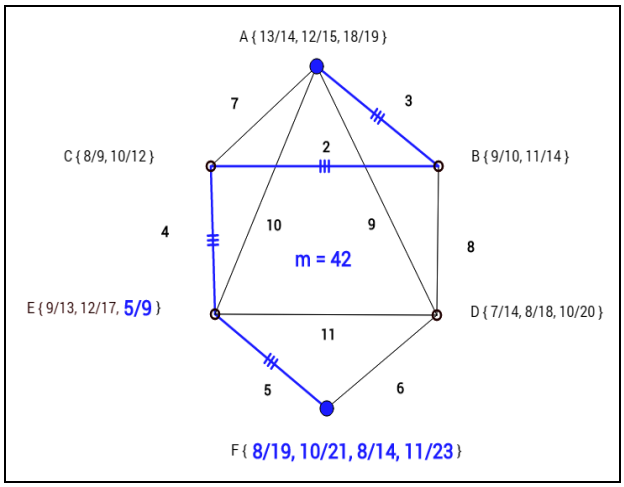

Fig. 2. $L_3^{path}(G^D)$, *priority 'a' (minimal edge weight)*

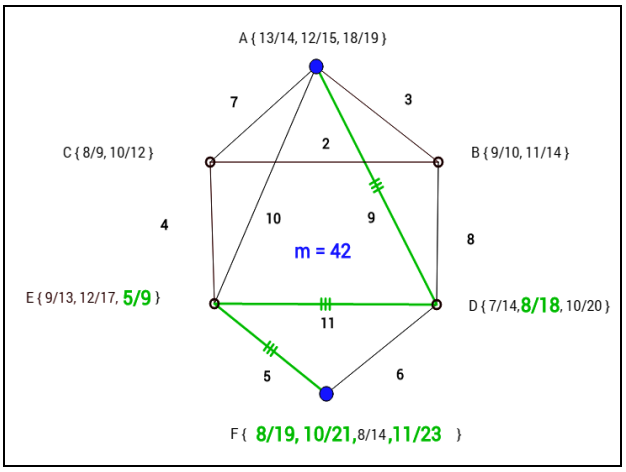

Fig. 3. $L_7^{path}(G^D)$ *, priority 'b' (maximal knapsack value)*

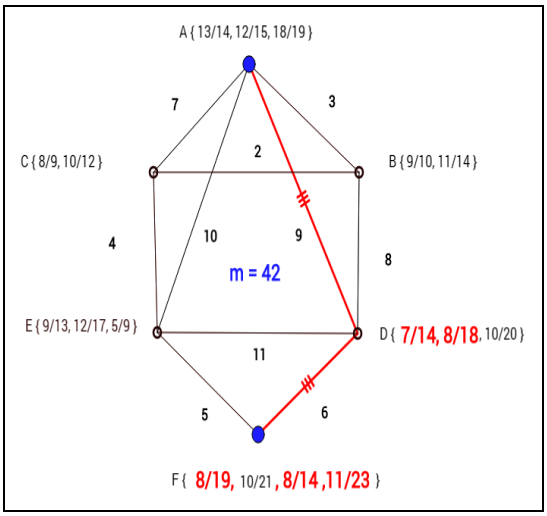

Fig. 4 $L_6^{path}(G^D)$ *, priority 'c' (maximal difference between knapsack value and edge weight)*

3.2 Algorithm correctness proof

3.2.1 Invariant

For Finding the most efficient paths in G^D algorithm, the invariant is "At iteration i, the $L_i^{path}(G^D)$ is the most efficient path, and has the maximal attribute value $(tkv_i - tew_i)$, or tkv_i or tew_i , decided by the priority choice. ".

3.2.2 Completeness/ Correctness

3.2.2.1 Initialization

For $i = 1$, the invariant is respected: in the first iteration, $L_1^{path}(G^D)$ is the most efficient path, and has the maximal attribute value $(tkv_1 - tew_1)$, or tkv_1 or tew_1, decided by the priority choice, since , trivially, it is the only path checked.

3.2.2.2 Maintenance

For $i = n$, Given $1 \le k \le n-1$, assuming the $L_k^{path}(G^D)$ is the most efficient path, and has the maximal attribute value $(tkv_k - tew_k)$, or tkv_k or tew_k. decided by the priority choice. Without the loss of generality we take tew_k. Iteration n inserts a new path- $L_n^{path}(G^D)$. Three possible cases for $L_n^{path}(G^D)$:

a. $tew_k < tew_n$, meaning the new maximal attribute path is tew_n.
b. $tew_k > tew_n$, meaning the maximal attribute path is preserves as tew_k .
c. $tew_k = tew_n$, meaning either a or b is acceptable.

Thus the invariant is preserved.

3.2.2.3 Termination

At the last iteration, Given $1 \le i \le size-1$ assuming the $L_i^{path}(G^D)$ is the most efficient path, we insert the last path and then take into consideration the three different cases above *(6.2.2)* , thus the $L_i^{path}(G^D)$ or the last path is the most efficient path, and has the maximal attribute value $(tkv_i - tew_i)$, or tkv_i or tew_i , decided by the priority choice. Hence the algorithm gives us the most efficient path.

3.3 More results and examples for G^D

In Tables 3 (vertices) and 4 (edges), we can see a another example for G^D – its item-vertices by their properties of value (v) and weight (w) and the edges by their weight (w^E), then the source and the target vertices are shown, and the knapsack weight limit (m).

Table 5 is ordered in the same manner as Table II described above, and displays a different, bigger graph G^D , with 8 vertices, 13 edges, and 13 different paths from the source vertex A to the target vertex H, and its results after performing the algorithm, and choosing the paths by the three different priorities.

Given the graph G^D, presented in Tables 3, 4 and 5, using A as the source vertex and H as the target vertex, and a knapsack weight limit of m=50, the first step is finding all of the paths between A and H.

An item is presented as its vertex and an index $1 \leq i \leq |V_i|$, e.g. the items of vertex A are marked A_1, A_2, A_3. An edge is marked by its source vertex and its target vertex, e.g. A-B.

There are 13 possible paths between A and H in G^D, L^{path} is presented in Table 5, 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 vertex- items that maximize the knapsack value- these are the chosen items of 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 Table 5, priority *'a'* (minimal edge weight)) is marked in blue, priority *'b'* (maximal knapsack value) is marked in green, and priority *'c'* (maximal difference between knapsack value and edge weight) is marked in red/orange.

Looking at the tables, we can now see that for priority *'a'*, the minimal edge weight achieved by L_{11}^{path} is 31, and that for priority *'b'*, the maximal knapsack value achieved by L_9^{path} is 103, and that for priority *'c'*, the maximal difference between the knapsack value and edge weight achieved by L_8^{path} is 70-32=38.

All of these results, including the ones presented in Table 1 and Figures 1-4, were achieved by an automated system, built in Java, for the purpose of this research.

The system gets the numeric inputs of the parameters (vertex-items and edges values, knapsack limit), and the graph's configuration (edges-items-vertices), and outputs the 3 different paths and their values, chosen by the 3 different priority types.

Table 3:G^D Scheme Example No.2-Vertices

G^D- *knapsack-item weight-independent graph*		
Vertices-items (V_i)		
$V(G^D)_i$	v	w
A_1	12	11
A_2	14	13
A_3	16	15
B_1	8	9
B_2	18	11
C_1	6	7
C_2	5	13
C_3	8	19
C_4	9	12
C_5	8	10
D_1	12	23
D_2	8	14
D_3	22	7
D_4	7	6
E_1	13	9
E_2	13	7
E_3	17	8
F_1	7	8
F_2	12	9
F_3	14	13
F_4	9	18
F_5	16	12
G_1	12	11
G_2	20	9
G_3	23	21
H_1	12	8
H_2	15	14
H_3	17	16
H_4	18	9

Table 4:G^D Scheme Example No.2-Edges

G^D- knapsack-item weight-independent graph		
Edges		
$E(G^D)$	w^E	
A-B	15	
B-C	21	
C-D	13	
A-C	9	
B-D	14	
D-E	12	
E-F	23	
F-G	19	
F-H	20	
G-H	18	
B-H	17	
C-H	22	
C-F	25	
m	**Source vertex**	**Target vertex**
50	A	H

Table 5: L^{PATH} for G^D Scheme Example No.2

L^{path} (G^D) For A→F, with m=42				
Paths				
No.	$V(L^{path})$	**Chosen items**	tkv_i	tew_i
1	A→B→C→D→E→F→G→H	D_3, E_1, E_2 , E_3, G_2, H_4	103	121
2	A→B→C→D→E→F →H	B_2, D_3, E_2 , E_3, H_1, H_4	100	104
3	A→B→C→H	A_2, B_1 , B_2, H_1, H_4	70	58
4	A→B→C→F→G→H	B_2, F_2 , F_5, G_2, H_4	84	98
5	A→B→C →F→H	B_2, F_2 , F_5, H_1, H_4	76	81
6	A→B→D→E→F→G→H	D_3, E_1, E_2 , E_3, G_2, H_4	103	101
7	A→B→D→E→F→ H	B_2, D_3, E_2 , E_3, H_1, H_4	100	84
8	A →B→H	A_2, B_1 , B_2, H_1, H_4	70	32
9	A→C→D→E→F→G→H	D_3, E_1 , E_2, E_3, G_2, H_4	103	94
10	A→C→D→E→F →H	D_3, E_1, E_2 , E_3, F_2, H_4	95	77
11	A →C→H	A_1, A_3, C_1 , H_1, H_4	64	31
12	A→C→ F→ G→H	A_1, F_2, F_5 , G_2, H_4	78	71
13	A→C→ F→H	A_1, F_2, F_5 , H_1, H_4	70	54
Chosen L_i^{path} by priority				
Priority	L_i^{path}			
a	**No.11**			
b	**No.9**			
c	**No.8**			

4 Conclusion- 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_2^{path} and L_7^{path}, both achieve tkv_i of 90. 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_7^{path}, achieving tew_i of 25 over L_2^{path}, achieving tew_i of 27. 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.

.

4.1 Ongoing work

For G^D, and the algorithm for finding the most efficient path between two vertices, there could be much more applications and expansions, like using the vertices labels as strings, or other data structures (a stack on each vertex, for example).

An implementation of a simpler version of the graph, in which the vertices are IoT (Internet of Things) objects, such as routers, other end-points or web services is currently being developed. A basic configuration example is shown in Fig.5 using A as the source vertex, that represents the client, and G as the target vertex that represents the host, whilst the other vertices represent the routers, and a knapsack weight limit of m=17 is given. The edges' weights represent the physical distance of the end-points, and for the vertices- the property of weight represents the data size (d_i), and the value represents the processing time (p_i).

Another possible problem inferred is a one in which a weight is added to each edge when an item is chosen from its source vertex, which can be handled in dynamic programming. A Java system was built for manufacturing the results shown in this paper, based on the algorithm described in part 4. Improving and expanding it as described here is a work in progress.

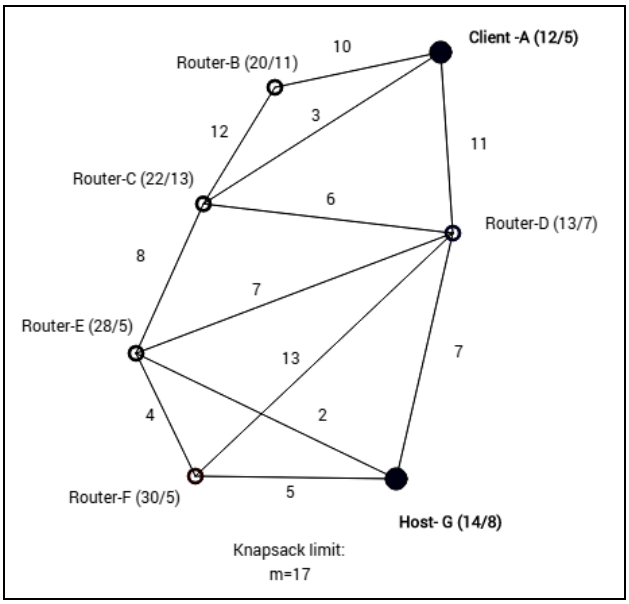

Fig.5 IoT graph-Vertices marked (pi/ di), source-A, target- G

References:

[1] Mathews, G. B. (1897). "On the partition of numbers" . Proceedings of the London Mathematical Society 28: 486–490.

[2] Abraham, Ittai; Fiat, Amos; Goldberg, Andrew V.; Werneck, Renato F. (2010) "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". ACM-SIAM Symposium on Discrete Algorithms, pages 782-793.

[3] Dantzig, George B. (1957). "Discrete-Variable Extremum Problems". Operations Research 5 (2): 266–288.doi:10.1287/opre.5.2.266.

[4] Pisinger ,D. (1993). " A Minimal Algorithm for the 0-1 Knapsack Problem " . Operations Research September-October 1997 45(5):758 doi: 10.1287/opre.45.5.758

[5] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" . Numerische Mathematik 1: 269–271.doi:10.1007/BF01386390.

[6] Mehlhorn, K., Sanders, P. (2008). "Algorithms and Data Structures: The Basic Toolbox" .199-200. Springer.

[7] Frieze, A. M. (1976). "Shortest path algorithms for knapsack type problems" . Mathematical Programming 11: 150. doi:10.1007/BF01580382

[8] Handler, Gabriel Y., Zang, Israel (1980), " A dual algorithm for the constrained shortest path problem", Networks- Wiley Subscription Services, Inc., A Wiley Company 10: 293-309, doi: 10.1002/net.3230100403

[9] Zhang, X., Huang, S., Hu, Y., Zhang, Y., Mahadevan, S., Deng, Y.: (2013) " Solving 0–1 knapsack problems based on amoeboid organism algorithm." Appl. Math. Comput. **219**, 9959–9970

[10] Voloch N. (2017) " Finding the most efficient paths between two vertices in a knapsack-item weighted graph ", International Journal of Advanced Computer Research (IJACR) Vol.7 Issue 28. http://dx.doi.org/10.19101/IJACR.2017.728003

[11] Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network" . Canadian Journal of Mathematics. 8: 399–404. doi:10.4153/CJM-1956-045-5.

[12] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". Journal of the ACM. Association for Computing Machinery. 19 (2): 248–264. doi:10.1145/321694.321699.

[13] Yefim Dinitz (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". Doklady Akademii nauk SSSR. 11: 1277–1280.