Solving Package Recommendation Problems with Item Relations and Variable Size

Christian Villavicencio, Silvia Schiaffino and J. Andrés Diaz Pace

ISISTAN Research Institute, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Campus Universitario, Argentina.
CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina.
{christian.villavicencio, silvia.schiaffino, andres.diazpace}@isistan.exa.unicen.edu.ar

Abstract. In this article, we explore an approach to solve the problem of recommending a package of items (each of them with a score and a cost) to a user. In our approach, we consider two types of relations between items: dependency and incompatibility; and we also consider that the size of the package is not fixed but cost-driven. To this end, adaptations of existing package recommendation algorithms are proposed. We have evaluated the proposed approach in a specific domain and obtained promising results.

Keywords

Recommendation Systems, Constraint-based Optimization, Prerequisites.

1 Introduction

Traditional recommendation systems deal with the problem of recommending items or sets of items to users by means of various techniques [1,9] In some domains, instead of recommending individual items, it is necessary to suggest *packages of items*, such as in tourism, or movie recommendation, among others. In most existing works [8,11], packages have a fixed size, using a value of k as the desired size of the package. This kind of problem is referred to as the *Top-K Package Problem* [3,5,8,10,11,12]. Furthermore, some approaches focus on certain aspects, such as the existence of prerequisites while recommending an item or, specially, a set (package) of items. For example, in [7] a prerequisite is defined as follows: "*A prerequisite of an item i is another item j that must be taken or consumed (done, watched, read, ...) in advance of i.*"

Existing works consider that each item has a score, in order to indicate how good it is for the users. Only a few approaches [11,12] take in account the cost of the items and a maximum cost limit (known as *budget*). To the best of our knowledge, none of them considers packages of variable size or packages whose items have a score and cost associated to them. In such cases, we can put as many items as the budget (cost limit) allows, without restricting a-priori the number of items for each package, while trying to maximize the package score.

In this context, we present an approach for package recommendation that considers a combination of the aforementioned factors. In particular, we recommend packages of items that have dependency (aka. prerequisites) and incompatibility relations, while trying to maximize a certain value, and also considering a maximum cost (budget). Our approach applies some of the techniques proposed by Xie's approach [11,12] (budget

and item incompatibility) together with those of Parameswaran [7]. To this end, we explore different adaptations of Xie´s and Parameswaran´s algorithms.

There are specific domains in which our approach can be beneficial. Examples of such domains include: i) selecting a list of movies to be watched [2,7] (considering that some movies might have to be watched in sequence); ii) searching for a travel package [6], given the initial and final destinations plus some requirements to be met by the package; and iii) determining what level of detail should the sections of a Software Architecture Documentation [4] (SAD) have, in function of the stakeholders that consume the SAD contents. In the movies domain [7], we can consider the movies to be recommended to a user in a package form as items with both a score and a cost. The score and cost for each item has to be given, for example, the score can be measured in terms of how much the user will be satisfied by the suggested package. The cost can be measured as the money necessary for buying or renting each movie, and therefore the budget will represent the maximum amount of money the user is willing to pay in order to watch some movies. Thus, the problem consists of recommending a package of movies that achieves the maximum user satisfaction while having a cost below the given budget.

The contribution of this work is a set of algorithms that combine existing techniques for dealing with constraint-based package recommendation problems. Our approach can be applied to problems that need to model dependency and incompatibility relations between items, or even certain restrictions regarding package size.

The rest of the article is organized as follows. Section 2 briefly covers the state of the art. In Section 3, we define the problem we aim to solve. Section 4 describes the algorithms that support the proposed approach. Section 5 presents some experimental results obtained with our approach. Finally, in Section 6, we give the conclusions of the article and discuss future work.

2 Related Work

The problem of recommending packages of items has been studied by several authors [3,5,7,10,8,12,11]. The closest approach to the problem we are trying to solve is [7]. These authors presented the item-package recommendation problem taking into account certain dependency relationships between the items, described as *prerequisite*s. However, their solution only solves problems with packages of fixed size.

In [10] the concept of *skyline* (which works similarly to the Pareto Frontier) is defined and then used to find the set of items (with fixed size) that maximizes a certain score while taking into account a maximum cost that cannot be exceeded. This approach relies on the concept of *Top-N Recommendation* proposed in [3], in order to create the mentioned skyline. This approach is also presented in [8] in conjunction with a variant of the problem: finding the top-k popular products focusing in customer preferences in order to make the recommendations.

The authors in [12] explore the idea of *composed recommendations* combined with *Top-N Recommendations* in order to generate the packages of items to recommend. The system finds the top-k packages of items with the highest total value such that each package has a total cost under a given budget and is also compatible. This way, the package size is not fixed but subjected to the budget restriction. Compatibility between items in a package models certain type of dependency relation between them. Xie´s approach is similar to ours, but it lacks prerequisite relationships between items. Finally,

in [11] an application of composite recommendation systems to the travel planning domain is discussed.

3 Problem Definition

As explained in Sections 1 and 2, the main idea of this article is to develop a solution to a variation of the problem of recommending packages of items, in which we try to maximize a score value while taking into account a cost limit. Following [12] problem specification, we initially have a set I of items and U of users, an active user $u \in U$ (the user the system is making recommendations for), and an item $t \in I$. Every item in the set has a *score* $s(t)$ and a *cost* $c(t)$.

A *package* P can be defined as a set of items that can be related or not. Since a package encompasses a set of items, it has two related values: *score* and *cost*, which are computed based on the members of the package, as shown in Eq. 1.

$$c(P) = \sum_{t \in P} c(t) \quad and \quad s(P) = \sum_{t \in P} s(t). \tag{1}$$

Then, we say that a package P is feasible according to a budget B iff $c(P) \leq B$.

3.1 Item relationships

When creating packages, some sort of relations between items can exist. In this work, we focus on two types of relations: dependency and incompatibility. The first type, shown in Fig. 1a, can be modeled using the prerequisite definition given by [7]:

Definition 1. *(Prerequisite) Given a directed graph $G(I,E)$ whose vertices $v \in I$ correspond to items, adirected edge $(v,w) \in E$ represents aprerequisite if item v needs to be taken (for a package) before item w. In addition, the constraint described by Eq. 2 must be satisfied to ensure that prerequisites are fulfilled:*

$$\forall v; w \in I \; : \; w \in P \wedge (v,w) \in E \Rightarrow v \in P \tag{2}$$

The second type of relation can be defined as follows:

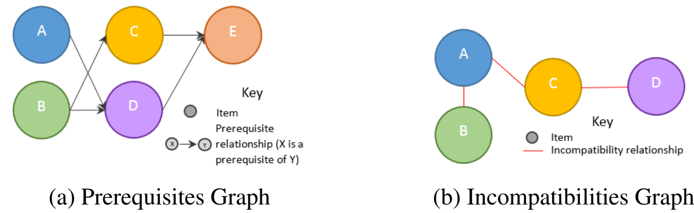
Definition 2. *(Item incompatibility) Two items v and w are incompatible if they cannot be within a package P at the same time.*

$$\forall v; w \in I \; : \; v \in P \Leftrightarrow w \notin P \; \wedge \; w \in P \Leftrightarrow v \notin P \tag{3}$$

This kind of relation can be modeled as a non-directed graph where vertices are items and edges between vertices represent incompatibility cases. An example of incompatible relations is shown in Fig. 1b in which we can see that $itemA$ is incompatible with $itemB$ and $itemC$, which is also incompatible with $itemD$.

Once incompatibility is defined, we say that an *inconsistency* exists within a package P if the number of incompatibilities for P ($iC(P)$), as described by Eq. 4, is greater than zero. If a package P is inconsistent, then the proposed solution is not valid.

$$iC(P) = number\,of\,incompatibilities\,between\,items\,in\,P \tag{4}$$



(a) Prerequisites Graph (b) Incompatibilities Graph

Fig. 1: Example of Prerequisites and Incompatibilities Graphs

3.2 Optimization formulation

Overall, taking into account the concepts explained before, the *package recommendation problem* can be stated as follows: given a certain budget B, find a feasible package A ($A \subseteq I$) that maximizes the score value $s(A)$, as expressed by Eq. 5, while covering all the existing prerequisites between items inside P and avoiding any inconsistency.

$$\begin{aligned} &maximize \quad s(A) \qquad while \quad c(A) \leq B \quad and \quad iC(A) = 0 \\ &\qquad with \quad A \subseteq I \quad and \quad s(A) = \sum_{t \in A} s(t) \quad and \quad c(A) = \sum_{t \in A} c(t) \end{aligned} \tag{5}$$

4 Proposed Approach

In order to implement algorithms that solve the problem formulation above, we decided to extend 3 heuristic algorithms proposed in [7] and modify one function which is used by two of them. In particular, we considered the following changes:

1. The function that computes the external set of a given package A, which is defined as *"the set of items that are not in A and can be potentially added to A without violating prerequisites"* (that is, either their prerequisites are already included in A or they have no prerequisites at all). This function is referred to as $external(A)$, and is used by both the *BF Pickings* and *TD Pickings* algorithms.
2. The *Breadth-first Pickings (BF Pickings)* algorithm, which in its original version firstly generates a package of size k (k being the package size limit) and then applies a refinement process over it. This refinement replaces the items with the lower score in the boundary of package A using the best item in $external(A)$. The boundary of a package A, $boundary(A)$, is the set of items that can be removed from A without violating the prerequisites of any other items in A).
3. The *Greedy-value Pickings (GV Pickings)* algorithm, which uses a priority queue Q to maintain sets of items. For each item in the graph of dependencies G they add to the queue a set containing that item and its prerequisites. For example: if item a depends on items b and c, and item d depends on item e, and b, d and e do not have any prerequisites, Q will contain the sets: a,b,c,d,e,b,d and e. Then, it loops trying to add the best set of the queue to A. If the set was added, the algorithm updates the sets in Q, if it was not added it changes the value of the set to zero (to avoid selecting it in the next step).
4. The *Top-down Pickings (TD Pickings)* algorithm, which in its original version generates the best size for k and then loops trying to add the prerequisites of the items to A. To do so, it removes items that are in the $boundary of A$, in order to "make space" for placing the prerequisites. If it wasn't possible to add the prerequisites of an item, the item is discarded.

In the sequel, we give a summary of our modifications to the algorithms.

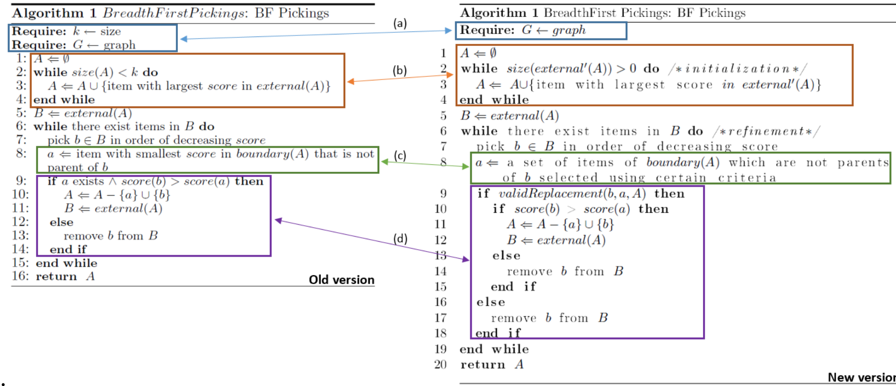


Fig. 2: BF Pickings Changes: Old vs New version

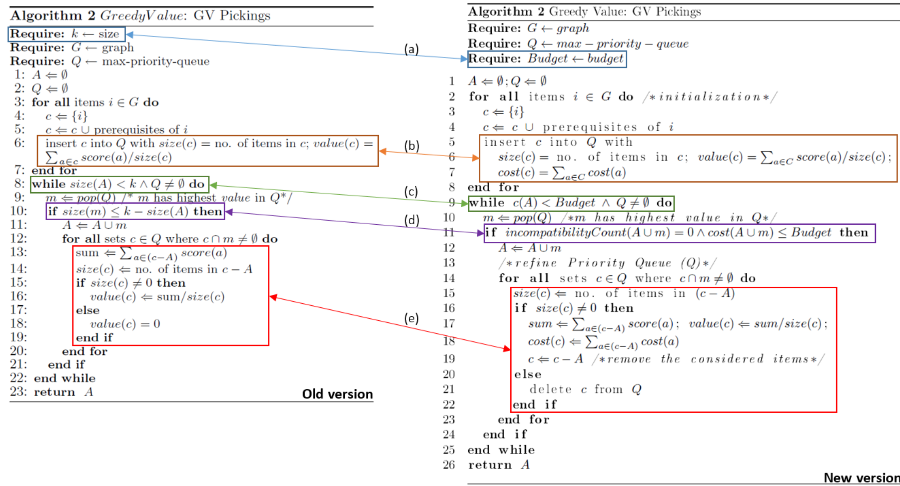


Fig. 3: GV Pickings Changes: Old vs New version

4.1 External function

Originally, $external(A)$ was defined as explained in Section 4. In order to adapt this function to our objectives, we must consider the information regarding to the cost of the items and the budget B when selecting those items that will be included in the *external set*, by checking that $cost(A) < B$. Additionally, we need to add a validation in order to asure the absence of incompatibility relations ($ic(A) = 0$). Both objectives can be achieved by adding an additional restriction to the original function.

4.2 Breadth-first Pickings (BF Pickings)

We modified the original BF Pickings algorithm along several points, as shown by Fig. 2. In (a) the new version does not need a k value anymore, as the package has not fixed size, so the initialization step of the original algorithm was removed. In (b) we changed the generation of the initialization data, we actually use the new external function (see Section 4.1). In (c) we changed the rules for selecting the items to be replaced. And finally, in (d) we replaced the $a\,exists$ check of the original version by a $validReplacement$ check, which is more complex than the one used before because

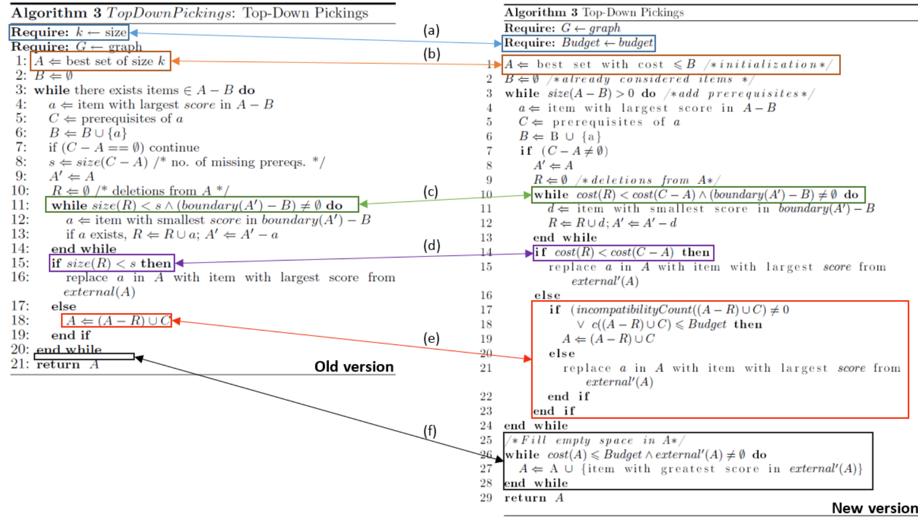


Fig. 4: TD Pickings Changes: Old vs New version

it does not only check that a exists and it is not empty but also checks if, after the replacement, the contraints added to the external function (see Section 4.1) are fulfilled.

4.3 Greedy-value Pickings (GV Pickings)

There are four points in the original *GV Pickings* algorithm in which it is necessary to make changes. These changes are depicted by the connected coloured boxes in Fig. 3. In (a), k variable initialization is replaced by budget variable initialization, as we do not need k anymore because the packages have not fixed size now. In (b) for every set c, we compute also its cost. In (c) we replaced the package size validation (that controls if the size after adding the set m exceeds the size limit k) with a validation that checks 2 conditions: i) if $A \cup m$ does not have inconsistencies, and ii) if its cost does not exceed the $Budget$. In (d) the loop condition is changed. In (e), when updating the sets of the priority queue for every set c with non-zero intersection with m , we updated c itself (by removing its items that are already included in A) and its score and cost . If the amount of items that are in $c - A$ is zero, then c is useless (because it will be empty after the update) and should be removed from Q.

4.4 Top down Pickings (TD Pickings)

For TD Pickings, we needed to make similar considerations as with BF Pickings and GV Pickings. Fig. 4 shows our main changes. Since the package size is not fixed anymore, in (a) and (b) we adjusted the algorithm to work with the budget restriction. In (b) the best set is generated by using an heuristic. In (c) we changed the loop restriction because there is no need to check the remaining space in the set, in terms of item counting. Now, we need to check if the cost of the removed items is lower than the cost of the prerequisites that are yet not in A $(C - A)$. The same argument is valid for (d). In (e), when trying to add the missing prerequisites of a to A, we need to check before the new restrictions (introduced with the modifications explained in Section 4.1). If any of the restrictions is not fulfilled, the item whose prerequisites could not be added should be replaced (removed from the set). Additionally, after the main loop, a “gap” can exist in

A. A gap is an space measured by comparing the cost of A and the budget B. Certainly, the constraint $cost(A) \leqslant B$ will be satisfied but $score(A)$ will not be the best score that could be achieved by using this technique. To solve this situation several approaches can be used. We decided to use a heuristic, which consists in filling the gap in the set by adding iterativetly items of $external'(A)$ to A, described (f) in Fig. 4.

5 Experimental Results

There are several problems that can be solved as constraint-based recommendation problems. Because of prior work in optimization of software architecture documentation [4], we decided to adapt the problem in order to solve it using the techniques proposed in this article.

Definition 3. *(SAD Optimization Problem) The Software Document Architecture (or SAD) consists of a set of documents, each one with a certain detail level. As completing all the documents (or sections of the SAD) takes considerable time and work, the SAD is instead completed iterativetly. Some of the documents are related by dependency relations (e.g. document A depends on document B if we need to complete B before A). Also, there are users interested in those documents (called stakeholders) and each one will be more or less satisfied if certain documents have the level of detail most appropriate for each stakeholder. The problem involves finding which documents should be modified in the next iteration (and with what detail level) while trying to maximize the satisfaction of the stakeholders, but taking into account that every modification has a cost and there is a cost limit for every iteration.*

In order to be able to solve the SAD optimization problem as a package recommendation one, we need to find a way to map the elements of the first problem to the ones of the second. This way: (a) $Item$: each SAD document can be linked to several items, each one representing a possible detail level that the document can have in the next version of the SAD. For example, if document A is in level 1, and the top level is 3, and we consider that the level cannot go down from one iteration to the next one, then three items will be linked to A when applying package recommendation: one for staying at level 1, say $item1$, one for going to level 2, say $item2$, and one for going from level 1 to 3, $item3$. (b) $ItemScore$: is the stakeholder satisfaction level respect to that item (this quantifies how much the stakeholders will be satisfied if the document takes the level indicated by the item). (c) $ItemCost$: it is the cost of taking the document (linked to the item) to the detail level indicated by the item. (d) $Budget$: it is the cost limit of the iteration. And (e) $ItemRelations$: dependency relations between SAD documents can be mapped to prerequisite relations of the proposed approach (e.g. document A depends on document B, so B is a prerequisite of A). Also, we can consider as $siblings$ to the items linked to the same document, and so, they cannot be in the solution at the same time, having by definition an incompatibility relationship.

After doing the mapping, we tested the approach over the SAD domain and obtained interesting results, as it can be seen in Fig. 5. The package recommendation algorithms obtained solutions with an acceptable stakeholder satisfaction level (when compared to previous results [4]). In terms of performance (measured by execution time), they also behaved well with an average execution time of 1 second. In terms of cost, all the solutions found had a cost equal to the budget. Finally, in terms of dependency satisfaction no dependency relationships were violated. The experiments were executed in a PC with: AMD FX-6300, 16 GB RAM and Windows 8.1 Pro.

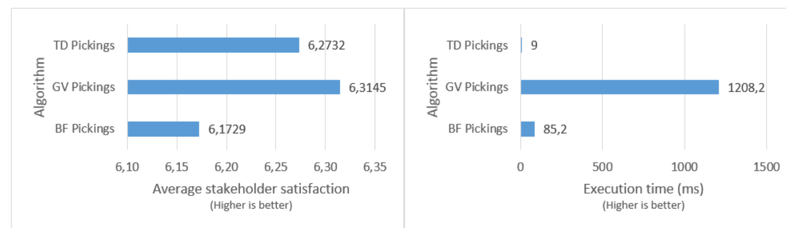


Fig. 5: Experimental Results

6 Conclusions

The main contribution of this article is a new proposal for solving constraint-based package recommendation problems, supporting both prerequisite and incompatibility relations between items and packages with variable size.

The results obtained so far (in a particular domain) are promising in terms of the stakeholder satisfaction level and the performance of the algorithms. However, we still need to evaluate the approach in other domains, such as tourism (travel planning) and movies. As future work, we also plan to: *i*) do a comparison between solutions generated with the proposed approach for the SAD domain and the solutions generated by prior optimization techniques [4] and group recommendation algorithms; *ii*) evaluate alternative formulations for the problem that can be tackled with SAT solvers [4] and *iii*) analize the theorical guarantees of our proposed algorithms.

Acknowledgements

This work has been partially funded by CONICET through project PIP 112-201101-00078 and ANPCyT through PICT 2011-0366.

References

1. Alexander Felfernig, Michael Jeran, G.N.F.R.S.R., Stettinger, M.: Basic Approaches in Recommendation Systems. Springer (2014)
2. Christensen, I. A. & Schiaffino, S.: Ratings estimation on group recommender systems. Revista Iberoamericana de Inteligencia Artificial (2012)
3. Deshpande, M., Karypis, G.: Item-based top-n recommendation algorithms. ACM Trans. Inf. Syst. (2004)
4. Diaz-Pace, J., Nicoletti, M., Schiaffino, S., Vidal, S.: Producing just enough documentation: The next sad version problem. In: Search-Based Software Engineering. Springer (2014)
5. Khabbaz, M., Xie, M., Lakshmanan, L.V.: Efficient algorithms for recommending top-k items and packages (2011)
6. Liu, Q., Ge, Y., Li, Z., Chen, E., Xiong, H.: Personalized travel package recommendation. In: IEEE 11th ICDM (2011)
7. Parameswaran, A.G., Garcia-Molina, H.: Recommendations with prerequisites. In: Proceedings of the 3th ACM RecSys (2009)
8. Peng, Y., Wong, R.C.W., Wan, Q.: Finding top-k preferable products. IEEE TKDE (2012)
9. Ricci, F., Rokach, L., Shapira, B., Kantor, P.B.: Recommender systems handbook. Springer (2011)
10. Wan, Q., Wong, R.C.W., Peng, Y.: Finding top-k profitable products. In: Proceedings of IEEE 27th ICDE (2011)
11. Xie, M., Lakshmanan, L.V.S., Wood, P.T.: Comprec-trip: A composite recommendation system for travel planning. In: Proceedings of IEEE 27th ICDE (2011)
12. Xie, M., Lakshmanan, L.V., Wood, P.T.: Breaking out of the box of recommendations: From items to packages. In: Proceedings of the 4th ACM RecSys (2010)