General Model for Requirements Prioritization and Assignment to Increments

Mohammad A. Asmaran
Prince Abdullah bin Ghazi Faculty of Information and Communication Technology,
Al-Balqa Applied University
Al-Salt, 19117, Jordan

ABSTRACT
Software development is a very consuming process as it consumes the development company resources, budget, and time. In some cases, there is a restriction on the workload that the company can perform. Moreover, in some cases software products are required in less time than estimated production time. To resolve such issues, software development is divided into increments that fit with those restrictions. This involves selecting a subset of requirements with higher priority. In this research a model is proposed to optimize the selection process of the requirements to be developed during an increment by maximizing returns and restricting other factors to the maximum restriction with dependency concern.

General Terms
Computer Applications, Algorithms, Software Engineering, Approximation.

Keywords
Agile, Requirements Prioritization, Maximum Independent Set.

1. INTRODUCTION
Software development process is an intensive process which consumes software company resources. In many cases, company resources such as development, power, and budgets are not enough to complete the entire development operations which leads the company to divide the project into releases (i.e. increments). Each increment is considered a step into completing the whole project. This process helps optimizing the usage of available resources to deliver the most urgent and effective requirements first for the customer in the early phases (increments). [1]

In order to determine software increments and their included requirement, each requirement should be assigned many values that represent its evaluation in terms of return, importance, and consumption. Return value should be maximized as much as possible with respect to Importance and consumption constraints. [1]

Assigned values that called weights, are determined from stack holders and project production resources (i.e. mainly developers). The weights are accumulated together with iterative process to reach the final values of the intended weights. [1]

To generalize the formula of weights calculation, it would be assumed **(n)** as the number of stakeholders and project production resources and **(m)** is the number of priority metrics. Each Stakeholder is assigned a specific weight in correspondence to each requirement, and priority metric. This would lead to general requirement prioritization formula of the j^{th} priority of the requirement R to be: [1]

$$R_j = \sum_{k=1}^{n}\sum_{i=1}^{m} W_{st_k(R_j,i)} * P_{st_k(R_j,i)} \qquad \text{Equation (1)}$$

In general, the following metrics represent some general metrics for any software development project and can vary across different projects or project phases (increments): [1]

- Importance:

Represents how much the evaluated requirement is important to the project.

- Penalty:

Represents the cost of delivering the requirement with delay.

- Cost:

Very huge term that represents the needed resources of different types to develop the requirement. the following is the main sub-factors of the cost:

  - Resources:
    Includes needed developers, hardware, stationary, and so on.
  - Budget:
    Simply it represents how much money is needed to cover all needed resources and other costing factors.
  - Market:
    Volume of market lost to deliver such requirement.

- Profit:

In the direct way it is the value of return – cost.

- Time:

How much time needed to deliver the requirement. In some cases, it is considered a part of the cost factor.

- Risk:

A very large term that represents different risk factors such as funding delay, requirement change, resource change, or underestimation.

- Volatility:

The probability of requirement removal in the future.

- Dependency:

The dependency of a requirement on another requirements to be developed first.

This research is organized by reviewing related works, followed by a description of the proposed model. Finally, formal validation by first order logic is used to validate the proposed model, which is followed by a proposal of future enhancement of the model.

2. RELATED WORKS
As discussed in the previous section, prioritizing process involves determining priority metrics and determine which

technique to use for weighting each metric. Finally, determine which requirement is suitable for the current increment and gain better return.

These metrics should be evaluated with a numerical value that reflects such metric. The main problem is that these metrics are hard to evaluate in term of personal evaluation of each party. So, many techniques are proposed to do so such as: [1]

2.1 Analytical Hierarchy Process (AHP)

One of the most accurate prioritization techniques, that compare every requirement with other requirements and assign it a value that relies between 1 and a maximum number (usually equals to the number of requirements) such as 5 in the example values illustrated in Table 1 that are assigned as the same technique used in the case study provided in [2]. Note that normalization could be done by dividing each score by the summation of all scores as calculated in the Table 2 based on the values provided in the example Table 1. This is a complex process suitable for small number of requirements or critical decisions [1, 2, 3, 4].

Table 1. AHP Example values.

	R-1	R-2	R-3	R-4	R-5
R-1	1	3	1/3	5	2
R-2	1/3	1	½	2	3
R-3	3	2	1	1/3	1/3
R-4	1/5	1/2	3	1	4
R-5	1/2	1/3	3	¼	1

Table 2. Normalized AHP values.

	R-1	R-2	R-3	R-4	R-5
R-1	0.1987	0.4390	0.0426	0.5825	0.1935
R-2	0.0662	0.1463	0.0638	0.2330	0.2903
R-3	0.5960	0.2927	0.1277	0.0388	0.0323
R-4	0.0397	0.0732	0.3830	0.1165	0.3871
R-5	0.0993	0.0488	0.3830	0.0291	0.0968

2.2 Cumulative Voting, the 100-Dollar Test

In this method, stakeholders have a maximum number of units to be used in priority assignments for all requirements. This maximum is usually but unlimited to 100 [1, 5].

2.3 Numerical Assignment (Grouping)

In this method three groups are defined, and stakeholders must distribute requirements to these groups. Note that to avoid unbalanced assignment of requirements to groups, maximum limit of each group is specified [1, 6].

2.4 Ranking

In this method stakeholders assign each requirement a unique rank. note that the top rank is 1 [1].

2.5 Top-Ten Requirements

Each stakeholder has to select top ten or (n) requirements without any order.

Note that a summary comparison of the prioritization techniques is shows in Table 3 [1].

Table 3. A Comparison of Prioritization Techniques.

Ranking	Complexity	Granularity
Highest	Top-Ten	AHP & 100-Dollar
↓	AHP	Ranking
	100-Dollar	
	Ranking	Grouping
Lowest	Grouping	Top-Ten

2.6 Prioritization Example

In this example, prioritization has been performed and calculated using the same technique provided in [1], a custom project has a set of requirements that must be filtered to meet a project specific cost constraint (In this example, it is assumed to be 70%). Table 4 represents an example list of priority techniques used for different project metrics. Tables 5 and 6 represent weights assigned by stakeholders, and the final priority value. Table 7 shows the selection of requirements according to cost metric. Table 8 represents better selection of requirements according to IP/cost metric that meets cost maximum limit.

Table 4. List of Priority Metrics & Prioritization Techniques

Metric	Technique	Participating Parties
Importance	100-Dollar	Customer Stakeholders
Penalty	AHP	Management
Cost	100-Dollar	Production Team (Development & Quality Control)

Table 5. Prioritization Results of Customer Stakeholders Importance Priority, P(RX) = RPS1 × WS1 + RPS2 × WS2 + RPS3 × WS3, where RP is the requirement priority, and W is the weight of the stakeholder.

Stakeholder	S1	S2	S3	Importance Priority
Requirement	**0.3**	**0.5**	**0.2**	
R1	0.1	0.13	0.2	0.14
R2	0.15	0.12	0.09	0.12
R3	0.3	0.09	0.07	0.15
R4	0.05	0.09	0.12	0.08

R5	0.07	0.03	0.06	0.05
R6	0.03	0.07	0.05	0.05
R7	0.11	0.16	0.18	0.15
R8	0.09	0.04	0.03	0.05
R9	0.06	0.06	0.1	0.07
R10	0.04	0.21	0.1	0.14
Total	**1**	**1**	**1**	**1**

Table 6. Descending Priority List Based on Importance and Penalty (IP). IP(RX) = RPI × WI + RPP × WP, where RP is the requirement priority, and W is the weight of Importance (I) and Penalty (P).

Priority Factor	**Importance**	**Penalty**	**IP**
Requirement	**0.65**	**0.35**	
R1	0.14	0.2	0.16
R2	0.12	0.3	0.18
R3	0.15	0.1	0.13
R4	0.08	0.1	0.09
R5	0.05	0.01	0.04
R6	0.05	0.03	0.04
R7	0.15	0.08	0.13
R8	0.05	0.09	0.06
R9	0.07	0.06	0.07
R10	0.14	0.03	0.10
Total	**1**	**1**	**1**

Table 7. Selected Requirements Based on IP and Cost.

	IP	**Cost**	**Selected**
R2	0.18	0.15	Yes
R1	0.16	0.13	Yes
R3	0.13	0.11	Yes
R7	0.13	0.1	Yes
R10	0.1	0.09	Yes
R4	0.09	0.05	Yes
R9	0.07	0.04	Yes
R8	0.06	0.3	No
R5	0.04	0.02	No
R6	0.04	0.01	No
Total	**1**	**1**	**0.67**

Table 8. Selected Requirements Based on Cost and IP/Cost Ratio.

	IP	**Cost**	**IP/Cost**	**Selected**
R6	0.04	0.01	4.00	Yes
R5	0.04	0.02	2.00	Yes
R4	0.09	0.05	1.80	Yes
R9	0.07	0.04	1.75	Yes
R7	0.13	0.1	1.30	Yes
R1	0.16	0.13	1.23	Yes
R2	0.18	0.15	1.20	Yes
R3	0.13	0.11	1.18	Yes
R10	0.1	0.09	1.11	Yes
R8	0.06	0.3	0.20	No
Total	**1**	**1**	**15.77**	**0.7**

Many researches are done to determine the accuracy of prioritization techniques such as AHP. In [7], AHP and CBRank techniques are compared and evaluated in term of three metrics, which are: ease of use, time-consumption and the accuracy on 23 real projects. In term of accuracy, AHP provides better outcomes over CBRank, but in the remaining factors CBRank outperforms AHP.

In [8], Prioritizing techniques are surveyed and evaluated to checkout their suitability for medium and large projects, which shows that for medium size projects it provides fine results in the opposite of large projects, which shows bad results.

In [9], a study of 11 successful software companies is done to determine practical prioritization techniques used. The study shows that priority grouping is the most used technique to minimize the number of requirements. The study shows the absence of customer from the prioritization process in many projects and the usage of subjective measures. It shows that quality requirements are not paid an attention from the decision makers as the focus should be.

In [10], two case companies are studied to evaluate prioritization techniques in practice. The study shows that the entire process was informal and suggests doing it iteratively with a systematic way that is difficult to achieve as customer preferences are not known.

In [11], a method for optimally allocating requirements to increments by assessing and optimizing the degree to which the ordering conflicts with stakeholder priorities within technical precedence constraints and uses genetic algorithms to find out optimal allocation.

In [12], suggests a model of requirement prioritization that relies on the known prioritization techniques. The model provides a relation to estimate factors from each other. Note that the main and first factor is cost estimation.

In [13], practical application of prioritization and business value delivery processes in eight software organizations has been investigated. The study revealed an important gap between the realities of the practitioners and the assumptions made in agile requirements engineering literature. It found that three explicit and fundamental assumptions of agile requirement prioritization approaches, as described in the agile literature on best practices, do not hold in all agile project contexts in the study. Those are: (i) the driving role of the client in the value creation process, (ii) the predominant position of business value as a main prioritization criterion, and (iii) the role of the prioritization process for project goal achievement.

3. PROPOSED MODEL

As listed in the literature, there are eight common factors (i.e. Importance, Penalty, Cost, Profit, Time, Risk, Volatility, and Dependency) to determine requirement priority. These factors are mainly subjective according to the project situation. The main problem is how to choose the best requirements to be implemented in a specific increment. So, the main question in this research is how to assign requirements to software development increment without violating increment constraints and maximizing return value of that increment. Proposed model is required to satisfy requirements constraints and keep track of requirements dependency too. So, the proposed model is intended to satisfy the followings:

1. Satisfy dependency constraints.
2. Satisfy weighted factors constraints.
3. Maximize total weighted factors.

3.1 Model Steps

1. Determining Priorities.
2. Workload Constraints.
3. Building Requirements Dependency Graph.
4. Map Requirements into undirected graph.

5. Merge restricted factors weights into one weight.
6. Merge the maximum needed weights into one weight according to their importance.
7. Apply (Modified Weight or any Approximated Maximum Independent Set algorithm in case of huge number of requirements [14, 15]) based Maximum Independent Set algorithm.

Where Maximum Independent Set Problem is described as the selection of maximum number of nodes in graph where any of these nodes are directly connected (or linked) to each other. This problem is solved by two types of algorithms: (i) exact algorithms, which finds the exact optimal solution where one of the most famous algorithms to do that is Modified Wilf algorithm, (ii) approximate algorithms, which tries to find the best approximated solution of the problem due to the nature of NP-Complete problems of consuming a lot of time to find its exact solution in case of more complicated inputs (i.e. graphs in case of Maximum Independent Set problem) [14]. Where mapping of real application such as scheduling problems to Maximum Independent Set problem is used by researchers such as in [15]. Which is used to find the maximum students classes schedule.

3.2 Requirements Priorities

Requirements priorities are the main factor of the process of creating increments and Requirements assignment to those increments. For the sake of optimal assignments, each requirement is assigned two types of weights, first type of weights is Maximized Weights that are intended to find their maximum gains such as Profit and Customers needs Priority (i.e. Importance). In the proposed model, such weights are project dependent, to illustrate project specific view of importance as it could differ between different projects. These weights should be accumulated in a single value according to Equation (2) that is derived according to Equation (1). In the following equation it is assumed that there are (n) priority values for each requirement. Each of those priorities should be multiplied by corresponding weight of such priority and accumulated in order to get overall maximized priority.

$$\text{Maximized J}^{\text{th}}\text{ Requirement Priority} = \sum_{k=1}^{n} W_{Priority_k(R_j)} * P_{k(R_j)} \quad \text{Equation (2)}$$

The second type of weights is constraint priorities that could not be merged into a single value, like the maximized one mentioned before. These weights represent the constraints of the increment such as cost, which is limited to assigned budget. Another example of such constraints is time factor, which is limited to increment time frame. Another example is Risk factor that could be a mixed value of many risk constraints, or more than one factor as they are differing in its nature or impact. Those mentioned examples usually could not be merged into one constraint value. These constraints should be respected in the model and should not be exceeded.

In this model, it is suggested to use workload constraint value in all the projects which represents production cost of the requirement. This factor could be computed using many ways such as CoCoMo model or developer's estimations. Regardless of the way of estimation, this estimation should be obtained and supported within available workload constraints.

3.3 Workload Constraints

As mentioned before, workload constraints should be embedded automatically in the model. This should be achieved by doing the following steps:

a. Work type definition (e.g. documentation, development, or validation, etc.).

b. Assignment of each resource to his/her type(s) of work that he/she can do (e.g. resource A could do development).

c. Each requirement is assigned (n) values that represent requirement workloads of the different types of work (e.g. Requirement 1 needs 10 units of time for documentation, 50 units for development, and 10 for validation).

d. According to available resources and time interval of increment, workload type constraint could be determined. (e.g. if there are 5 developers working within 10 business days according to 8 hours working hours, development workload constraint would be 10*8*5 = 400 hours assuming that time unit is an hour).

By doing such classification, workload could be dynamically assigned according to different types of working resources jobs and levels. This is used to map practical issue of real development that assigns some critical tasks to well skilled development resources (i.e. senior ones).

3.4 Building Requirements Dependency Graph

As mentioned before, workload constraints should be embedded automatically in the model. This should be achieved by doing the following steps:

As mentioned before there are two types of weights to deal with for optimal requirement assignment to an increment, which are: (i) maximized weights and (ii) constraint weights. There is more critical factor to deal with for optimal assignments that represents the need of implementing requirement to be able to do another requirement. (e.g. you can't obtain general ledger account balance if journal voucher is not implemented). This dependency is very important for optimal assignment and should be determined. In this model, it is suggested to build dependency graph as the following:

1. Define each requirement labeled inputs and labeled outputs (e.g. a requirement to compute Z=X+Y has two inputs X, and Y and one output Z).

2. If a requirement modifies an input to produce it as an output (e.g. x = x*y) the input would be x version (i) and the output would be x version (i+1).

3. If an output of a requirement is used as an input to another one, this would be considered as dependency (e.g. if R1 produces X version (1) and R2 inputs X version (1), R2 would be considered to be dependent on R1).

3.5 Undirected Graph Construction & Dependency Mapping

As mentioned before, the first step is to map dependencies into undirected graph. The mapping process is performed according to the following rules:

Rule 1: If the requirements are restricted to be developed together, they will be mapped into one requirement node. Weights of the new node are the summation of all nodes.

Rule 2: If two requirements are sequentially dependent so that one requirement development is dependent on the development of the other requirement. They would be mapped into two requirements nodes. The first one is the original dependent requirement; the second one is a new merged requirement with the summation weights of the merged requirements. These nodes would be connected with a link.

Rule 3: If the sequence contains more than two nodes, they will be mapped into the same number of requirement nodes. Each requirement node weights are the summation of all dependent nodes. Suppose that $R_0 \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \ldots. R_N$, $W_k(NR_i) = \sum_{j=0}^{i} R_j W_k$. Note that resulted nodes are fully connected.

Rule 4: Branched requirements dependencies are divided into multiple sequential requirements. Each sequential dependency line is mapped as in case 3. Note that nodes should be fully connected.

Rule 5: If any independent nodes restricted weights summation exceeds restriction value, a link is added to connect them together.

Rule 6: If any node restricted weight exceeds its corresponding requirement restriction, node will be removed from the resulted graph.

In Figure 1, an example of requirements dependency graph that denotes that R2 is dependent on R1. R3 and R4 are dependent on R2.

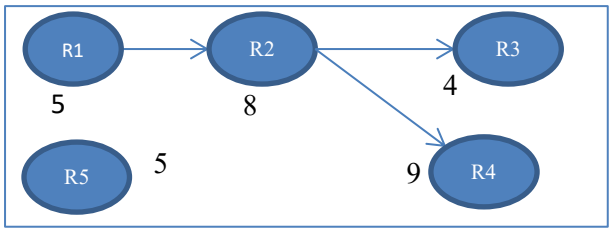

Fig 1: Sample Dependency Graph.

According to the conversion rules, figure 2 dependency would be converted into the undirected graph shown in figure 3. If the maximum weight is assumed to be 15, the resulted graph would be as shown in figure 3 which adds a link between R5 and (R1,2) as there are a total weight of 18 which is greater than 15 (weight constraint weight)..

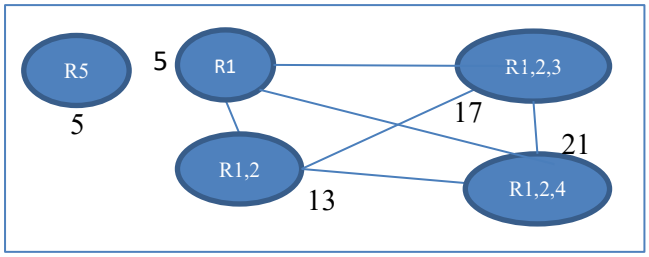

Fig 2: Mapped Graph.

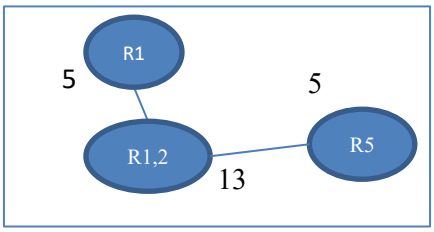

Fig 3: Mapped Graph with Weights Restriction.

3.6 Merge restricted factors weights into one weight

Restriction exists on many factors such as cost and risk factors. These factors should be merged into one weight as they are needed in the knapsack algorithm to find out maximum value according to restricted single weight. In the requirement model, restriction is not limited to single weight. This encourages merging the weights into single weight. This is done by the following:

$$\text{Merged Weight} = \sum_{i=1}^{n} \max(f_{i-1}) * f_i \qquad \text{Equation(3)}$$

For example, if risk varies between 0 and 10 and cost weights varies between 0 and 100, merged weight would be (Cost Weight) + 100*(Risk Weight) or 10* (Cost Weight) + (Risk Weight). So, if requirement cost weight is 70 and risk weight is 3, merged weight would be either 370 or 703.

3.7 Increment Requirements Assignment

Once the nodes are linked in a graph. To assign requirements to an increment, Maximum Independent Set algorithm is applied on the resulted undirected graph described earlier. This would find maximum non-conflicting nodes. Maximum Independent Set algorithm is modified to consider maximum Weight in its operation. Moreover, knapsack algorithm is merged inside to find out maximum weight according to restrictions.

Consider figure 4 as an example requirements graph. The selected requirement nodes would be R1, R5, and R6 without restriction consideration. If the restriction is considered to be 21, the selection would be either R1 and R5 or R1 and R6.

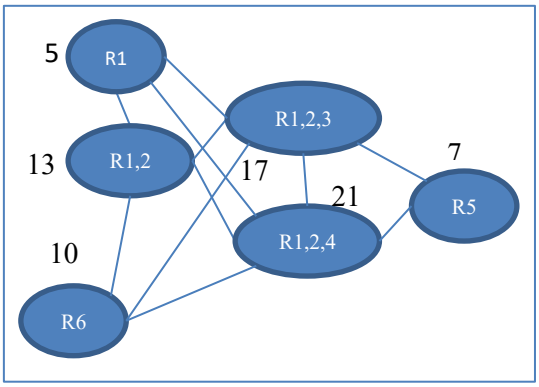

Fig 4: Requirements Graph.

So, overall requirements assignment algorithm would be summarized as in the following algorithm pseudo code while figure 5 illustrates the overall flowchart of the proposed model steps, note that detailed pseudo code is appended in appendix A:

```
generateBestIncrementAssignments(Vector requirements)
return Vector
        buildDependencies(requirements)
        mergeConstraints(requirements)
        requirements =
generateNodesCombinations(requirements)
        requirements = findMaximum(requirements)
    return requirements
end generateBestIncrementAssignments
```

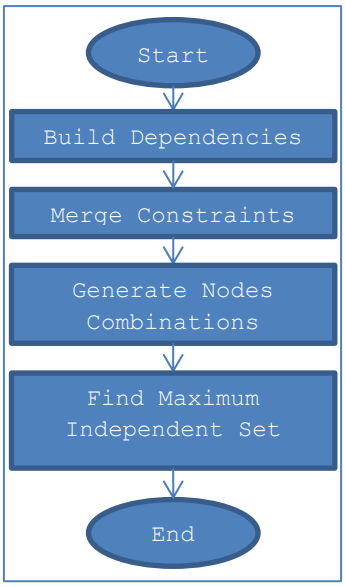

Fig 5: Proposed Model Flowchart.

4. VALIDATION

To validate this new model, the expected results of the intended model should be specified. Expected results would be the maximum weighted requirements that satisfy project constraints, and no requirement is assigned without the assignment of its pre-requirement if it is dependent on another requirement.

As explained in model steps, after obtaining requirement weights and details, directed graph is constructed according to the inputs and outputs of the requirements. Then, dependent nodes are merged into one node that represents the summation of the weights and constraints of the two nodes. Original nodes are not valid anymore and replaced by new merged ones. So, after the execution of this step assuming (S) is the resulted set of new merged requirements:

a. ∀ R1,R2 ϵ S → ¬ (dependent(R1,R2) ∨ dependent(R1,R2))
b. ∀ R1,R2 ϵ S ∧ share_sub_requirements(R1,R2) → link(R1,R2)
c. ∀ R1,R2 ϵ S ∧ ¬ share_sub_requirements(R1,R2) →¬ link(R1,R2)
d. ∀ R1,R2 ϵ S ∧ ¬ link(R1,R2) → ¬ share_sub_requirements(R1,R2)

Next step is to apply modified Wilf algorithm to find maximum independent set of the undirected graph. So, after the execution of this step is a set S' which represents maximum unconnected set of the undirected graph:

a. ∀ R1,R2 ϵ S' → ¬link(R1,R2)
b. final_solution(S') → ¬ ∃ S'' ∧ solution(S'') ∧ size(S') >= size(S'')

Modified Wilf is modified to compare solution on the accumulated weight of the tested solution rather than size of the solution. So, second condition of this step would be modified to:

final_solution(S') → ¬ ∃ S'' ∧ solution(S'') ∧ weights_sum(S') >= weights_sum(S'')

Constraints should not be violated. So, tested solution is passed to dynamic knapsack algorithm to pick maximum value with respect to constrained value which satisfies the following conditions:

a. final_solution(S') → ¬ ∃ S'' ∧ solution(S'') ∧ weights_sum(S') >= weights_sum(S'')
b. ∀ S' solution(S') → constraints_sum(S')<= MAX_CONSTRAINTS_SUM

When merging all above conditions, the following condition would represent the result:

∀ R1,R2 ϵ S' ∧ final_solution(S') → ¬ ∃ S'' ∧ solution(S'') ∧ weights_sum(S') >= weights_sum(S'') ∧ constraints_sum(S')<= MAX_CONSTRAINTS_SUM ∧ ¬link(R1,R2)

While the following conditions hold:

a. ∀ R1,R2 ϵ S ∧ ¬ link(R1,R2) → ¬ share_sub_requirements(R1,R2)
b. ∀ R1,R2 ϵ S → ¬ (dependent(R1,R2) ∨ dependent(R1,R2))

This would leads to the final condition:

∀ R1,R2 ϵ S' ∧ final_solution(S') → ¬ ∃ S'' ∧ solution(S'') ∧ weights_sum(S') >= weights_sum(S'') ∧ constraints_sum(S')<= MAX_CONSTRAINTS_SUM ∧ ¬link(R1,R2) ∧ ¬ share_sub_requirements(R1,R2) ∧ ¬ (dependent(R1,R2) ∨ dependent(R1,R2))

Final condition stated that final solution would be the maximum weights summation without violating constraints and no shared requirements are duplicated. Moreover, it states that there is no existence of dependent requirements in the graph. Dependency is satisfied and achieved by merging dependent requirements into one new requirement.

5. CONCLUSIONS

As described in the previous section, it is proved that proposed model would assign best selection of requirements according to the rules of respecting restrictions and maximizing benefits without any violation to dependencies. Proposed model involves performing a very complex process in term of processing as it is fully dependent on two complex algorithms (i.e. Knapsack and Maximum Independent Set algorithms). High complexity could not be a problem because of the absence of time restrictions on the expected output time interval while approximation algorithms can be used rather than exact ones.

Proposed model is suitable to be extended and integrated with traditional prioritization techniques so that conflicting priorities would be avoided which will help avoiding development obstacles due to missing dependencies or time conflicting requirements.

6. FUTURE WORK

The main challenge that is not implemented in the proposed model is multi-skill resources. The challenge is that these resources workloads can be assigned to different types of workloads which has to be assigned in the optimal way to maximize total maximized weight of the increment. This situation could be valid if work types would be classified according to resource level of experience. For example, if work types of development are classified to be junior and senior developers. If a resource A is assigned a senior developer work type, it should be assigned junior developer work type too because he is capable to do junior work too. The main challenge here is how to classify workload in such situation, a percentage or full assignment to a single type of work. If percentage is considered, how to obtain optimal work type percentages to achieve best results.

7. REFERENCES

[1] Berander Patrik, Andrews Anneliese (2005), Requirements Prioritization. In Engineering and Managing Software Requirements (pp. 69-94). Heidelberg, Berlin: Springer.

[2] Nancy Mead (2013), Requirements Prioritization Case Study Using AHP, Software Engineering Institute "Carnegie Mellon University",https://insights.sei.cmu.edu/library/requirements-prioritization-case-study-using-ahp/.

[3] Taherdoost, Hamed. (2017). Decision Making Using the Analytic Hierarchy Process (AHP); A Step by Step Approach. International Journal of Economics and Management Systems.

[4] Tavana, M., Soltanifar, M. & Santos-Arteaga, F.J. Analytical hierarchy process: revolution and evolution. Ann Oper Res 326, 879–907 (2023). https://doi.org/10.1007/s10479-021-04432-2.

[5] P. Chatzipetrou, L. Angelis, P. Rovegård and C. Wohlin, "Prioritization of Issues and Requirements by Cumulative Voting: A Compositional Data Analysis Framework," 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, Lille, France, 2010, pp. 361-370, doi: 10.1109/SEAA.2010.35.

[6] Aneesa Rida Asghar, Shahid Nazir Bhatti, Atika Tabassum and S Asim Ali Shah, “The Impact of Analytical Assessment of Requirements Prioritization Models: An Empirical Study” International Journal of Advanced Computer Science and Applications(IJACSA), 8(2), 2017. http://dx.doi.org/10.14569/IJACSA.2017.080240.

[7] Anna Perini, Filippo Ricca b, Angelo Susi,Tool-supported requirements prioritization: Comparing the AHP and CBRank methods, Information and Software Technology, 51 (2009), pp. 1021–1032.

[8] Qiao Ma, 2009,The Effectiveness of Requirements Prioritization Techniques for a Medium to Large Number of Requirements: A Systematic Literature Review, M.Sc. Thesis, Auckland University of Technology: NEW ZEALAND.

[9] Richard Berntsson Svensson, Tony Gorschek, Björn Regnell, Richard Torkar, Ali Shahrokni, Robert Feldt,Aybuke Aurum, Prioritization of Quality Requirements: State of Practice in Eleven Companies, in the proceedings of IEEE 19th International Requirements Engineering Conference, 2011, pp. 69-78.

[10] Laura Lehtola, Marjo Kauppinen, Sari Kujala,Requirements Prioritization Challenges in Practice, Product Focused Software Process Improvement, Volume 3009, 2004, pp 497-508.

[11] D. Greer, G. Ruhe, Software release planning: an evolutionary and iterative approach, Information and Software Technology, 46, 2004, pp. 243–253.

[12] Andrea Herrmann, Maya Daneva, Requirements Prioritization Based on Benefit and Cost Prediction:An Agenda for Future Research, 16th IEEE International Requirements Engineering Conference, 2008, pp. 125-134.

[13] Zornitza Racheva, Maya Daneva, Klaas Sikkel,Roel Wieringa, Andrea Herrmann, Do We Know Enough about Requirements Prioritization in Agile Projects: Insights from a Case Study, in the proceedings of Requirements Engineering Conference (RE), 2010 18th IEEE International, 2010, pp. 147 – 156.

[14] Mohammad A. Asmaran, Ahmad A. Sharieh, Basel A. Mahafzah (2019), "Chemical Reaction Optimization Algorithm to Find Maximum Independent Set in a Graph", International Journal of Advanced Computer Science and Applications (IJACSA), pp. 76-91, Volume 10, Issue 9.

[15] Ahmad A. Sharieh ,Mohammad A. Asmaran ,Basel A. Mahafzah , (2020) " Generating Class Scheduling without Conflict based on Maximum Independent Set " , International Journal of Advances in Science, Engineering and Technology(IJASEAT) , pp. 77-83, Volume-8, Issue-4

8. APPENDIX A

```
class SystemVariable{
        String name
        int version
}
class RequirementNode{
        String name
        int constraintWeights[]
        int weight
        int constraintWeight
        Vector dependentNodes
        Vector dependeningNodes
        Vector inputs
        Vector outputs
        Vector links
}
buildDependencies(Vector requirements)
        for each requirement in requirements
                for each input in requirement.inputs
                        for each mainRequirement in
requirements
                                if
mainRequirement.outputs contains input

        mainRequirement.dependeningNodes.add(requirement)

        requirement.dependentNodes.add(mainRequirement)
                                end if
                        end for
                end for
        end for
end buildDependencies
generateNodesCombinations(Vector requirements) returns Vector
        Vector combinations
        for each requirement in requirements
                Vector childCombinations =
generateNodesCombinations(requirement.dependeningNodes))
                for each childCombination in
childCombinations
                        childCombination.name =
requirement.name + "," + childCombination.name
                        childCombination.weight =
childCombination.weight + requirement.weight
                        for i=0 to
childCombination.constraintWeights.length

        childCombination.constraintWeights[i] =
childCombination.constraintWeights[i] +
requirement.constraintWeights[i]
                        end for

        combinations.add(childCombination)
                end for
                childCombinations.add(requirement)
                for i=0 to childCombinations.size()
```

```
                                RequirementNode node =
childCombinations.get(i)
                                for j=i to childCombinations.size()
                                        RequirementNode
otherNode = childCombinations.get(j)

        node.links.add(otherNode)

        otherNode.links.add(node)
                                end for
                        end for
                        combinations.addAll(childCombinations)
        end for
        return combinations
end generateNodesCombinations
mergeConstraints(Vector requirements)
        for each requirement in requirements
                requirement.constraintWeight = 0
                for i=0 to requirement.constraintWeights.length
                        factor = 1
                        for j = 0 to j<i
                                factor =
factor*MAX_CONSTRAINT[i]
                        end for
                        requirement.constraintWeight =
requirement.constraintWeight + requirement.constraintWeights[i] *
factor
                end for
        end for
end mergeConstraints
knapsack(Vector requirements,int constraint) returns Vector
        int values[requirements.size+1][constraint+1]
        Vector solution
        boolean keep[requirements.size()+1][constraint+1]
        for i=1 to values.length
                RequirementNode node = requirements.get(i-1)
                for w=0 to values[i].length
                        if
weightCombinations(node.constraintWeight)<=weightCombinations(
w) and values[i-1][w]<node.weight+values[i-1][w-
node.constraintWeight]

        values[i][w]=node.weight+values[i-1][w-
node.constraintWeight]
                                keep[i][w] = true
                        else
                        values[i][w]=values[i-1][w]
                        keep[i][w] = false
                        end if
                end for
        end for

        int maxConstraintWeight = constraint
        for i=values.length-1 to 0
                if keep[i][maxConstraintWeight]
                        RequirementNode node =
requirements.get(i-1)
                        solution.add(node)
                maxConstraintWeight=maxConstraintWeight-
node.constraintWeight
                end if
        end for
        return solution
end knapsack
Vector max
execute(Vector nodes,Vector solution,Node start)
        if isDisconnected(nodes)
                solution.clear()

        solution.addAll(knapsack(nodes,MAX_CONSTRAINT))
                if getWeight(max)< getWeight(solution)
                    max.clear()
                    max.addAll(solution)
                end if
                        return
                end if

                Vector all
                Vector graph1
                Vector graph2

                all.addAll(nodes)
                RequirementNode node = start
                if node=null
                        node=selectNode(all)
                end if
                all.remove(node)
                Vector neighbors
                neighbors.addAll(node.links())
                execute(all,graph1,null)

                for i=0 to neighbors.size
                    RequirementNode temp = neighbors.get(i)
                    all.remove(temp)
                end for

                execute(all,graph2,null)
                if checkConstraints(node,graph2)
                graph2.add(node)
                end if

                if getWeight(graph1)>getWeight(graph2)
                        solution.addAll(graph1)
                        return
                end if
                solution.clear()

                solution.addAll(graph2)
end execute
selectNode(Vector nodes) returns RequirementNode
        RequirementNode node = nodes.get(0)
        for i=1 to nodes.size
                if
getLinksCount(nodes.get(i),nodes)>getLinksCount(node,nodes)
                        node = nodes.get(i)
        return node
end selectNode
getLinksCount(Node node,Vector nodes) returns integer
        int count = node.weight
        for i=0 to node.links.size
                if nodes.contains(node.links.get(i))
                        count=count+1
                end if
        end for
        return count
end getLinksCount
isDisconnected(Vector nodes) returns boolean
        for i=0 to nodes.size
                for j=0 to nodes.get(i).links.size
                        if
nodes.contains(nodes.get(i).links.get(j)
                        return false
                        end if
                end for
        end for
        return true
end isDisconnected
getWeight(Vector nodes) returns integer
        int sum = 0
for i=0 to nodes.size
                RequirementNode node = nodes.get(i)
                sum=sum+node.weight
        end for
        return sum
end getWeight
checkConstraints(Node node,Vector nodes) returns boolean
        int consSums[CONSTRAINTS.length]
        for i=0 to CONSTRAINTS.length
```

```
                consSums[i] = node.constraintWeights[i]
                for j=0 to nodes.size
                        RequirementNode temp =
nodes.get(j)
                        consSums[i] =consSums[i] +
temp.constraintWeights[i]
                end for
                if consSums[i]>CONSTRAINTS[i]
                        return false
                end if
        end for
        return true
end checkConstraints
findMaximum(Vector nodes) returns Vector
        Vector maxSolution
        execute(nodes,maxSolution,null)
        return maxSolution
end findMaximum
```