Preferences in a Knowledge-Based Recommender for Product Configuration

Bachelorarbeit

von
Cedric-John Martens
Matr.-Nr.: 780954

June 16, 2022

Erstbegutachtung: Prof. Dr. Torsten Schaub
Zweitbegutachtung: M.Sc. Seemran Mishra

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln (z. B. Nachschlagewerke oder Internet) angefertigt habe. Alle Stellen der Arbeit, die ich aus diesen Quellen und Hilfsmitteln dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht und im Literaturverzeichnis aufgefuhrt. Weiterhin versi- ¨ chere ich, dass weder ich noch andere diese Arbeit weder in der vorliegenden noch in einer mehr oder weniger abgewandelten Form als Leistungsnachweise in einer anderen Veranstaltung bereits verwendet haben oder noch verwenden werden. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Datum:

Unterschrift

Table of Contents

Abstract. This thesis implements a recommendation system based on Answer Set Programming (ASP). Preferences that can express different user needs are discussed and implemented in ASP. Additionally, different recommender features such as similarity and ranking of solutions are discussed and implemented using preferences. 2 ASP implementations are given, one based off of weights and priorities and one based off of pareto optimality. Their usage, strengths and weaknesses are then shown on some examples.

1 Introduction

Configuration systems allow for the creation of individualized solutions and are an important tool in mass customization [8]. With them, the user is no longer restricted to simply choosing a product from a predetermined list. Instead, a user can configure a product with the help of a configurator by specifying feature requirements and desired attributes based on a set of available options [8]. At the same time, the process can provide companies with different benefits such as better management of product variety, improved product quality or simplified order acquisition [16]. Examples of industries where product configuration is applied are computers, vehicles, bicycles, medical systems, financial systems or telecommunication systems [8, 16].

The huge amount of options provided by a configuration system can leave the user overwhelmed. Even experts can easily fall back to configurations that they are used to and overlook better suited alternatives [15]. This gives rise to a need for systems to help recommend relevant options and solutions.

In most cases, recommender systems work on a predetermined set of products and their attributes. In the context of product configuration an explicit representation of all products is not feasible due to the amount of possible combinations of components [5, 15].

One well-known type of recommender system is the collaborative type. These systems use the ratings on items by other users in order to calculate a utility score [1]. Content-based methods, meanwhile, try to estimate the utility of an item based on the ratings a user has given on other items [1]. While hybrid systems exist to negate their individual weaknesses, they are still inflexible, meaning the user can not customize recommendations by themselves, and they suffer from the cold start problem, meaning a sufficient amount of interactions by a user or with an item need to be performed before a meaningful recommendation can be performed [1].

Knowledge-based recommender systems use domain knowledge and requirements provided by the user in order to recommend relevant options [1,6]. Knowledge-based techniques can address some of the drawbacks of content-based or collaborative recommender systems. However, knowledge-based systems require domain knowledge in order to work [1]. In combination with the aforementioned hybrid recommender systems, the different approaches are highly complementary [1, 2]. Examples of knowledge-based recommenders being used would be for restaurant or movie recommendation [2].

ASP is a declarative modeling language that can be used for product configuration. In fact, one of its first applications was in product configuration [14]. While recommender systems using ASP exist [11], they are domain dependent or require knowledge about ASP to use. The problem of recommender systems for product configuration has remained largely untouched by ASP.

The goal of this thesis is to create a knowledge-based recommender system that uses user-defined preferences as a basis for recommendation. The recommender should be applicable independently from the domain and require no expertise in ASP to specify preferences. The recommender system will be built on top

of a domain-independent configuration system in ASP, which will be briefly introduced in chapter 2, along with a background on recommenders and ASP. In chapter 3 I will establish a list of preference types that are required to cover different user preferences and explain a list of recommender features that can be implemented with preferences. In chapters 4 I will show two implementations of these preferences in ASP, once in the form of weak-constraints and once with the help of asprin, a language extension for the clingo solver. Afterwards, in chapter 5 I will show an exemplary recommendation process and show how the encodings work based on the example.

2 Background

2.1 Product configuration

The recommender is built on top of the configuration system introduced in [4]. The system separates domain-knowledge and configuration-knowledge. This separation allows for the configuration-knowledge to be reused in the context of different domains. This section gives an overview of the aspects of product configuration that are relevant to product recommendation. For the remainder of the thesis I will primarily use the bike domain example introduced in [4]. A visual representation of the domain can be seen in Figure 1.

The configuration problem consists of a set of components with a domain of component types. Examples of components would be the frame or the wheels of a bike. Additionally, f1, f2, and f3 represent different component types for the frame that can be used. Each component type has a set of attributes with their respective domains. Examples of attributes are price, colour or material. On top of that, a configuration problem features a set of constraints regarding attribute assignment, partonomy, requirements, incompatibility and user requirements. The details of these constraints can be largely ignored in the context of product recommendation.

A configuration problem can be defined as a set of components with attributes, a set of domains for each component attribute, and a set of constraints [13]. A configuration solution is an assignment of values to component attributes in the configuration problem from their domains such that the constraints are satisfied [13]. In [4, 13] a configuration is represented as a set of 3-tuples consisting of components, attributes and values. In the configurator used here, a configuration is given as a set of tuples `assign(C, A, V)` where `C` is a component, `A` is an attribute of the component and `V` is the value chosen for the attribute.

The configurator allows for the specification of user requirements which are represented as constraints. However, the requirements only allow for the inclusion or exclusion of a component or a specific attribute value. However, these user requirements do not cover all the possibilities a knowledge-based recommender requires. For example, they can not encode a preference that requires the price to be below a specific limit.

2.2 Product recommendation

Recommendation is the process of suggesting relevant items to the user. Due to the very high number of options available during the configuration process, users can easily be overwhelmed by the amount of choices available to them [8]. Some form of guidance is required to help the user navigate the solution space. The goal of recommendation is to help the user during the configuration process by filtering solutions that are irrelevant to the user and ranking solutions that are relevant to the user. For this purpose different recommendation methods can be used.

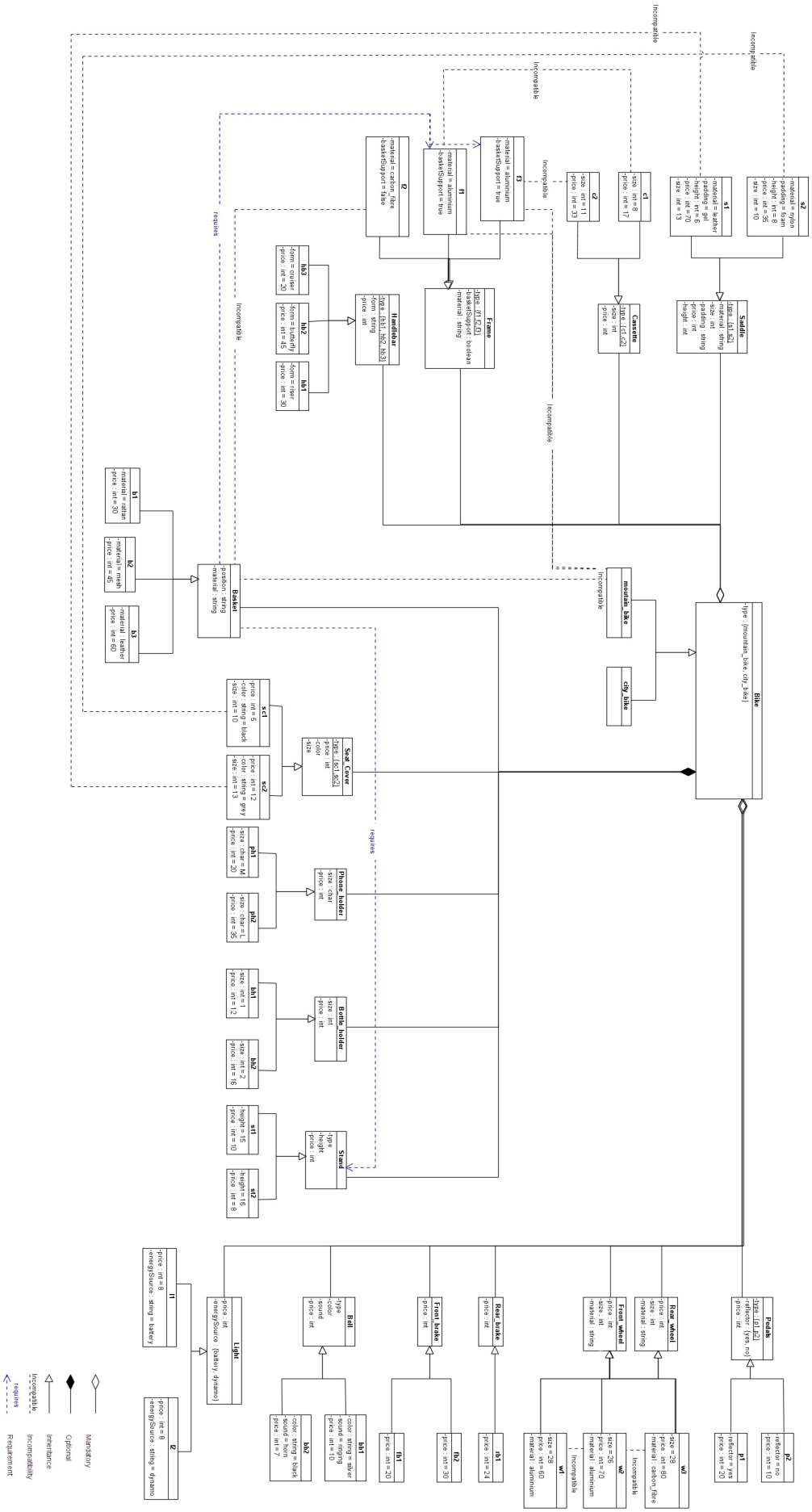

Fig. 1. Bike configuration in UML [4]

2.2.1 Content-based recommendation *uses information about previous ratings and choices by the user in order to rank similar solutions.* Content-based recommendation systems can use algorithms from information retrieval and machine learning [1].

Content-based systems require information about items to be provided in order to calculate similarity. Additionally, content-based systems are inflexible in that changing user tastes may not be recognized immediately and information about a user needs to be gathered before educated choices can be made (cold start problem) [1, 8].

2.2.2 Collaborative recommendation *uses information about previous ratings and choices by similar users in order to rank possible solutions.* Similar to content-based recommendation, algorithms from content retrieval and machine learning are used in collaborative systems [1].

Collaborative Systems require many ratings by different users. Items with few ratings end up getting recommended very rarely. Additionally, collaborative recommenders suffer from the cold start problem as well. New items start without any ratings and information about users needs to be gathered in order to find similar users [1, 8].

Content-based and collaborative recommendation techniques are often combined in *hybrid recommender systems*. While they can complement each other in order to reduce their weaknesses, some issues like the cold start problem still remain.

2.2.3 Knowledge-based recommendation *uses reasoning and domain knowledge in order to produce relevant solutions.* It allows for direct interaction with the user by allowing them to state their requirements and ideas either directly or through dialogue in order to influence the recommendation process.

A simple representation of a knowledge-based recommendation process is shown in Figure 2.

As opposed to their content-based and collaborative counterparts, knowledge-based recommender systems do not require information about user interaction first and can quickly adjust to changing preferences and requirements during the configuration process. However, the domain knowledge needs to be acquired and maintained [1, 2]. If the knowledge-base is not properly maintained components may end up with wrong domains for attributes or under-described, meaning that information about certain attributes is missing.

A knowledge-based recommendation approach that can be used as a basis are FindMe systems [2,3]. The FindMe systems have 2 modes of retrieval: *similarity-finding* and *tweaking* [2]. In the case of similarity-finding, the user can select an item and other items with similar attributes will be recommended to them. The similarity-finding mode can be adapted to product configuration by trying to keep each component similar across solutions.

In the case of tweaking, the user can request a change to a reference solution. The user can pick a solution they like and then, for example, request a "Nicer"

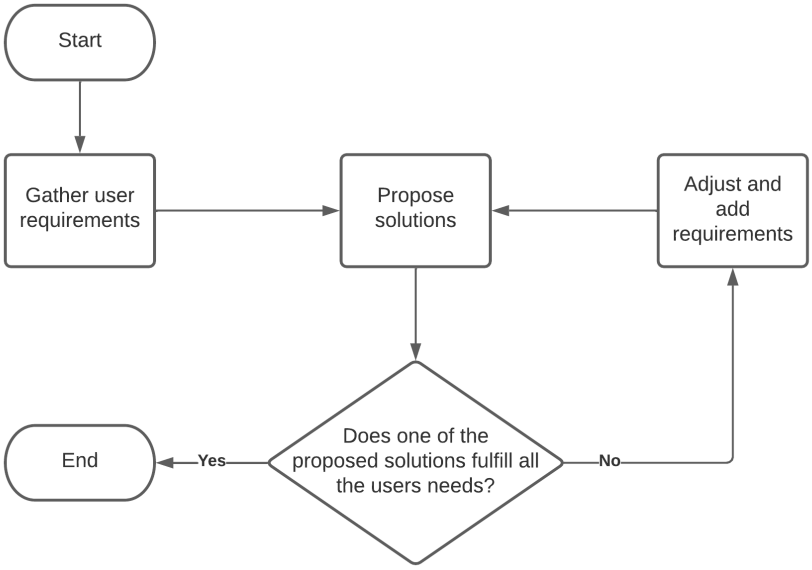

Fig. 2. A simple representation of the knowledge-based recommendation process

option [2]. The recommender would then look for similar options that score higher in their "niceness" value.

During retrieval different attributes are assigned a priority and then sorted using a list of buckets. The highest priority metric is applied first and those products that score highest in that metric are kept. Then the next highest priority metric is applied and only the best scoring products are kept, and so on. This is repeated until all metrics have been applied or only 1 product is left. The last bucket of products then contains the recommendable products.

This approach also gives an implicit ranking. All answers in the last bucket fulfill the given metrics to the best possible degree. Afterwards, the answers can then be ranked according to content-based or collaborative metrics. At worst, this would equal a random selection among the optimal answers [2].

Kenwood is a FindMe system that works on a configuration problem. However, while the user could create their own configurations as their starting point, the recommendations would be selected from a list of preconfigured solutions. The general ideas of tweaking and similarity-finding in knowledge-based recommendation systems can still be used as a reference for features that a recommender for product configuration should provide.

2.3 Answer Set Programming

ASP is a declarative modeling language which uses logic programming to provide models, called answer sets. An answer set program consist of rules of the form:

$$a \texttt{ :- } L_1, \ldots, L_n.$$

Where a is an atom and $L_1, \ldots, L_n$ are literals or default negations of literals, that is an atom preceded by `not`. A fact is expressed by only keeping a, while an integrity constraint is expressed by skipping a. The solution of answer set programs constitutes answer sets, whose semantics is given by [10].

Clingo allows for the expression of optimization criteria via weak constraints or optimization statements. Optimization statements extend ASP to search for optimal answer sets. A minimize statement has the form

$$\texttt{\#minimize}\{w@p, t_1, \ldots, t_n : L_1, \ldots, L_n\}.$$

with the weight w, an optional priority p, and tuples of terms t_i and L_i. An answer set is optimal if the sum of weights is minimal among all answer sets of the given program [9]. If priorities are given, optimization statements with a higher priority are compared first. A maximize statement works the same as a minimize statement, but w gets negated instead.

2.3.1 Asprin provides a framework for clingo for handling preferences in ASP and expands the options for optimization. The less(weight) preference type in asprin works similarly to minimization statements in clingo. A minimization statement as seen above can be translated into a less(weight) preference as follows [9]:

$$\texttt{\#preference}(\texttt{minimize}, \texttt{less}(\texttt{weight}))\{w, t_1, \ldots, t_n : L_1, \ldots, L_n\}.$$

The preferences in asprin do not include a priority. An alternative which I use in this paper is the comparison of preferences in a pareto preference, which tries to establish pareto optimality across preferences.

Pareto optimality can be used in multi-objective optimization problems if no criterion can be clearly preferred over another one. The approach in FindMe systems works under the assumption that it is possible to tell which preferences should have a higher priority and which preferences should have a lower priority. *In asprin a solution s is pareto optimal iff no other solution s' exists that fulfills at least one preference to a higher degree and all other preferences to at least the same degree as it does.*

2.4 Recommendation in ASP

ASP has been successfully applied to the problem of product configuration before [4, 7]. While knowledge-based product recommenders have been made with ASP

before [11], an ASP-based recommender system for product configuration has yet to be implemented. The goal of an ASP-based recommender would be to produce a set of models that are most relevant based on the preferences given.

In the following sections of this thesis I will set a basis for a knowledge-based recommender system in ASP.

3 Preferences in a Product Recommendation System

The user requirements introduced in [4] work as clear rules that remove any conflicting configurations from the set of possible answers. This means that in the case of a conflict with the knowledge domain, other user requirements or requirements set by the owner, no possible solutions would be left over. User requirements are unable to encode some requests, like a request for the lowest price option available, without having to manually search and select the cheapest option for each component.

Preferences are a set of rules based on which the configuration solutions are ranked. As opposed to user requests, preferences do not filter the set of possible solutions. Instead, they provide a ranking for solutions, by incurring a cost whenever a preference is not satisfied.

An optimal solution is a solution that fulfills all preferences to the best possible degree. The exact definition of this optimality is left to the implementation.

Preferences can be differentiated on the basis of who sets them, whether it is the consumer or user, the seller or owner, or the system. Preferences set by the owner could be to give all components the same colour for a solution to appear more attractive or to maximize profit margin. An example for system preferences would be diversity preferences, which are introduced later in this chapter. Preferences set by the owner or the system would usually be assigned a lower priority than preferences set by the user.

In this section different types of preferences will be presented.

3.1 User Preferences

I start by creating a list of required features for preferences that can handle direct user requests like "It should cost no more than 400€" or "The colour should be blue". In the configurator, a configuration is given as a list of 3-tuples containing the component C, the attribute A of the component and the value V assigned to the attribute from its domain. As a result, the different preference types work on the basis of these same 3 variables.

The first preference types are for *the inclusion and exclusion of an optional component.* An example would be a request for a basket on a bike.

A preference type for the inclusion or exclusion of an attribute is not needed, since all component types available for a component, share the same set of attributes.

On non-numeric attribute values, a preference type *for or against a specific attribute value* can be used. An example would be a preference for a specific colour or against a possible feature of a component. This preference type can also be used on numeric attribute domains, but its usefulness in this context is limited. However, there are additional options for numeric attribute domains.

On numeric attribute values, a preference type for choosing *the minimum or maximum possible value on an attribute* can be used. A very common example would be the search for the lowest cost option. Users might also want to minimize or maximize the sum of all values of an attribute type across all components.

Again, instead of looking for the lowest cost on just a component, a user might want to look for the lowest total cost configuration.

To expand on this idea, a preference type is needed, that can express a desire *for a value to be above or below a specific threshold.* An example of its usage would be a customer setting an upper limit on how much they want to spend. By combining an upper and a lower threshold, the desire for a value to be between 2 values can be expressed. Users might also want to apply this preference type to the sum of all values of an attribute type across all components.

The last preference types for staying above or below a threshold can be split up into 2 subcategories. If costs are incurred for not fulfilling the preference, the costs could either be fixed or depend on the distance to the threshold. The latter case would effectively turn the preference into a minimization or maximization type, if it is unfulfillable.

To summarize the different preference types:

- **Comp**: Request for the inclusion of a specific component
- **NegComp**: Request for the exclusion of a specific component
- **Attr**: Request for an attribute to have a specific value
- **NegAttr**: Request for an attribute to not have a specific value
- **Min**: Request for an attribute to have the lowest possible value
- **Max**: Request for an attribute to have the highest possible value
- **LE**: Request for an attribute to have a value less or equal a specific value
  - **LEW**: Request for an attribute to have a value less or equal a specific value. If the value is greater than the limit a fixed cost is incurred
- **GE**: Request for an attribute to have a value greater or equal a specific value
  - **GEW**: Request for an attribute to have a value greater or equal a specific value. If the value is smaller than the limit a fixed cost is incurred

3.2 Priority and weight

Each preference is also assigned a priority and a weight independently from its type. *The highest priority preferences are evaluated first.* All optimal solutions are then evaluated according to the next highest priority preferences. This process is repeated until no more preferences remain.

The weight is the cost that is incurred, if a preference is not fulfilled. If 2 preferences have the same priority but different weights, the preference with the higher weight will be preferred in case of a contradiction, but can be ignored if multiple priorities of the same priority are in conflict with it. If 2 preferences with the same priority and weight are contradictory, then any solution that fulfills either preference is considered optimal.

Additionally, suitable alternatives could be expressed by assigning a lower priority or weight to the alternative. An example of such a case would be if leather as a material would be too expensive. In that case, a suitable alternative might be faux leather. [2] already expresses the need to assign priorities to different metrics or preferences.

3.3 Recommender Features

Using the preference types introduced above, several features that can be used in recommendation systems can be expressed.

3.3.1 Ranking of solutions can be performed based on how many preferences are satisfied. If a preference is not fulfilled it incurs a cost. Solutions can then be ranked based on how high their accumulated costs are across the different priorities of preferences. Multiple solutions can share the same rank. If too many solutions share the highest rank, collaborative or content-based recommendation methods may be used to create a ranking among the optimal solutions.

3.3.2 Similarity *is a metric that describes how similar two configurations are when compared to one another.*

During browsing a user might come across a solution that is close to what they want but requires some further tweaking. In that case the user might want to express that they like that solution and wish to use it as a reference solution in order to tweak it or keep on browsing through similar solutions.

In order to encode similarity, the reference solution can be re-encoded in the form of preferences. It should be made sure that every component that was included in the reference configuration is included in the similar solutions as well. At the same time, any optional component that was not included in the reference solution should be excluded. The Comp and Negcomp preferences serve that purpose, respectively.

Next, all components need to have similar attribute values to the ones in our reference configuration. For non-numeric values Attr preferences can be used for this purpose.

For numeric values, a similar value should be as close as possible to the value in the reference configuration. Using LE and GE in combination, it can be expressed that any value that is not equal to the reference value should incur a cost, depending on how far away it is from the reference value. Alternatively, a fixed cost can be incurred if the preference is not satisfied. The latter is less restrictive, which can leave a greater pool of solutions for other, lower-priority preferences that are applied afterwards.

At the end a constraint is added, forbidding the exact same configuration to be picked again, as naturally a configuration would be most similar to itself.

For a normal product recommender this would give us similar products, that can then be recommended to the user. However, in a configuration problem many recommended configurations would differ in only 1 or 2 component types and use the exact same component types for all other components. This is because, while the exact same configuration can not appear again, it is still possible to pick almost the exact same configuration. An example of this happening can be seen in Figure 3. Solutions that barely differ like this are of little to no interest to the user. If the user is browsing for a similar solution, they may still expect enough differences to call a recommended solution new. One way would be to simply

limit the amount of components and attributes that are compared. Alternatively, preferences that require diversity can be used.

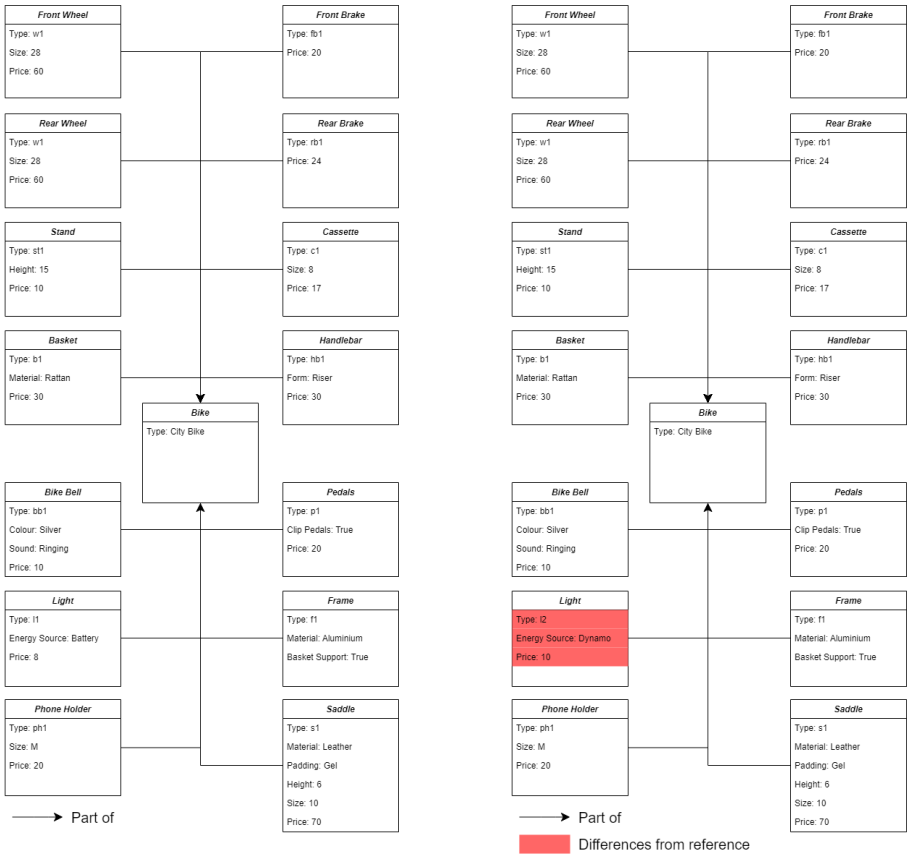

Fig. 3. Searching for similar solutions without diversity preferences: Only the lights are different.

3.3.3 Diversity

is a metric that describes how diverse two configurations are when compared to one another. It effectively acts as a negation of similarity.

While our new solution is as similar to our reference as possible, a new problem occurs now where our solutions need a certain degree of diversity. To solve this, I propose diversity preferences that express the desire for certain aspects of the reference solution to be avoided. Using NegAttr it can be expressed that the component type for either all or some components should be different but keep similar attribute values, if possible. This way it is possible to get similar solutions that the user can still identify as new solutions.

Diversity preferences could also be applied to attributes that might be deemed less important and where no clear preference from the user has been established. These attributes would in turn need to be excluded from the similarity-finding process.

The usage of diversity preference does not have to be limited to the context of similarity. If a user is simply browsing options, it is important to make sure that the options they are given are diverse enough for them to actually feel like different options.

3.3.4 Tweaking *is the process of adjusting similarity preferences to fit specific needs.*

A user might like a proposed solution but want some changes to be made. They could tweak it by simply requesting a basket to be included or even by requesting more vague changes. For example, if a bike should be "nicer" this can entail many different changes, for example the materials used or a separate aesthetics-score. At the same time, it might be necessary to adjust the price limit to adjust for those changes. The exact changes that a complex tweak like this may entail depend on the problem domain and can have different possible implementations.

In order to encode tweaking first similarity preferences need to be added. Then the preferences expressing the tweak are added with a higher priority.

While tweaking, diversity preferences are not included. The user likes the solution and knows what they want to change about it.

4 Product Recommendation in ASP

I have created 2 implementations of preferences in clingo. The first implementation uses clingo's built-in weak-constraints, or optimization statements. The second implementation uses the asprin framework, an extension for clingo.

The goal of these encodings is to *find a set of models (configuration solutions) that satisfy and optimize the preferences to the best possible degree.*

In order to encode a preference it is necessary to specify the preference type, the component and attribute it should be applied to, the value that should be applied and priority and weight to better control the recommender. This means a preference is given as follows:

```
prefer(T, (C, A, V), (P, W))
```

where `T` is the type of the preference, `C` is a component, `A` is an attribute of the component, `V` is the value chosen for the attribute, `P` is the priority of the preference and `W` is the weight the preference has.

where T is the type of the preference, C is a component, A is an attribute of the component, V is the value chosen for the attribute, P is the priority of the preference and W is the weight the preference has.

4.1 Optimization Statements

The preferences now need to be implemented in a functional encoding in ASP. In order to do this I use optimization statements or weak constraints, which have been introduced in subsection 2.3. In the encoding

```
#minimize{W@P,C : component(C),prefer(negcomp,(C,_,_),(P,W))}
```

means a cost of W at priority P is incurred, if the component C is assigned and a preference of the type NegComp for C exists. The full encoding of all preference types can be found in Listing 1.

Listing 1. Encoding of preferences using optimization statements

```
1  total(A, T) :- #sum{V, C, A : assign(C, A, V)} = T,
       property_val(_,A,_).
2
3  % min/max over all components
4  #minimize{V@P, A, C : assign(C, A, _), score(C, A, V), prefer(
       min, (all, A, _), (P, W))}.
5  #maximize{V@P, A, C : assign(C, A, _), score(C, A, V), prefer(
       max, (all, A, _), (P, W))}.
6
7  % min/max over specific component
8  #minimize{V@P, A, C : assign(C, A, _), score(C, A, V), prefer(
       min, (C, A, _), (P, W))}.
9  #maximize{V@P, A, C : assign(C, A, _), score(C, A, V), prefer(
       max, (C, A, _), (P, W))}.
```

```
10
11
12 % prefer/disprefer specific attribute
13 #maximize{W@P, V, A, C : assign(C, A, V), prefer(attr, (all, A
       , V), (P, W))}.
14 #maximize{W@P, V, A, C : assign(C, A, V), prefer(attr, (C, A,
       V), (P, W))}.
15
16 #minimize{W@P, V, A, C : assign(C, A, V), prefer(negattr, (all
       , A, V), (P, W))}.
17 #minimize{W@P, V, A, C : assign(C, A, V), prefer(negattr, (C,
       A, V), (P, W))}.
18
19
20 % prefer/disprefer components
21 #maximize{W@P, C : component(C), prefer(comp, (C, _, _), (P, W
       ))}.
22 #minimize{W@P, C : component(C), prefer(negcomp, (C, _, _), (P
       , W))}.
23
24
25 % prefer a value greater equal or less equal for all/specific
       components
26 % a fixed cost is incured if the preference is not statisfied
27 #minimize{W@P, A : total(A, T), prefer(leR, (all, A, X),(P, W)
       ), T >= X}.
28 #minimize{W@P, A : total(A, T), prefer(geR, (all, A, X), (P, W
       )), T <= X}.
29
30 #minimize{W@P, A : assign(C, A, V), prefer(leR, (C, A, X), (P,
        W)), V >= X}.
31 #minimize{W@P, A : assign(C, A, V), prefer(geR, (C, A, X), (P,
        W)), V <= X}.
32
33 % prefer a value greater equal or less equal for all/specific
       components
34 % the cost incured depends on how far away the actual value is
        from the threshold
35 #minimize{X-T@P, A : total(A, T), prefer(le, (all, A, X),(P, W
       )), T >= X}.
36 #minimize{T-X@P, A : total(A, T), prefer(ge, (all, A, X), (P,
       W)), T <= X}.
37
38 #minimize{X-V@P, A : assign(C, A, V), prefer(le, (C, A, X), (P
       , W)), V >= X}.
39 #minimize{V-X@P, A : assign(C, A, V), prefer(ge, (C, A, X), (P
       , W)), V <= X}.
```

In order to apply a preference to the total of a numeric value across all components 'all' can be used as the value of C. This is necessary if, for example,

the total price should be within a certain range. Preferences for specific non-numeric attributes via attr or negattr can also be applied to all components at once this way.

The priorities in the optimization statements work in the way that is described in subsection 3.2 and [2]. First, all optimal solutions for all statements with the highest priority are gathered and then the process is repeated with the remaining statements with the next highest priority preferences. Once only 1 solution remains or all preferences have been evaluated, a set of at least 1 optimal answer has been found.

4.1.1 Diverse Models [12] can be computed using the idea of diverse solutions introduced in subsubsection 3.3.3. This is useful if the user is simply browsing for options. Once again, all the solutions shown to the user should be different enough to be seen as a new solution and not simply a small modification to another recommended solution.

There are 2 possible approaches to this. The configurator could be expanded to give multiple solutions in one answer and then optimize for diversity between all solutions in an answer. This process is described in [12]. The second option uses multishot solving. Whenever an optimal solution is found, diversity preferences describing the last found solution are added and the next solution is searched. The previous diversity preferences are kept. This process is then repeated until the desired number of solutions has been reached.

I will use the second approach based on multishot solving as the first option adds more requirements to the configurator.

4.2 Asprin

The system asprin provides a framework for clingo for handling preferences in ASP. The preferences in ASP work similar to minimization statements in clingo. Consider the previous optimization statement for the NegComp preference.

```
#minimize{W@P,C : component(C),prefer(negcomp,(C,_,_),(P,W))}
```

This optimization statement can be expressed as a asprin preference of the less(weight) type as follows [9]:

```
#preference(comp(P), less(weight)) {
    W,C :: component(C),domain(C,A,V),
           prefer(negcomp,(C,A,V),(P,W))
} : prefer((negcomp), (_, _, _), (P, _)).
```

In the same way, all other preferences that can be expressed as an optimization statement can also be expressed as a preference of the less(weight) type in asprin. The only difference is in the role of the priority tuple given in the preference because the preferences in asprin are evaluated on a pareto front, as described in 2.3.1. An optimal solution in asprin is a solution that is pareto optimal in regards to the preferences.

Listing 2. Encoding of preferences using asprin

```
#include "defaults.lp".

total(A, T) :- #sum{V, C, A : assign(C, A, V)} = T,
    property_val(_,A,_).

#preference(weight(P), less(weight))
{
    S, A, C :: assign(C, A, S), prefer(min, (all, A, V), (P, W
        )); % #minimize statements
    -S, A, C :: assign(C, A, S), prefer(max, (all, A, V), (P,
        W)); % #maximize statements
    S, A, C :: assign(C, A, S), prefer(min, (C, A, V), (P, W))
        ;
    -S, A, C :: assign(C, A, S), prefer(max, (C, A, V), (P, W)
        )
} : prefer((min;max), (_, _, _), (P, _)).


#preference(boundry(P), less(weight))
{
    W, A :: total(A, T), prefer(leR, (all, A, X), (P, W)), T
        >= X;
    W, A :: total(A, T), prefer(geR, (all, A, X), (P, W)), T
        <= X;
    W, A, C :: assign(C, A, T), prefer(leR, (C, A, X), (P, W))
        , T >= X;
    W, A, C :: assign(C, A, T), prefer(geR, (C, A, X), (P, W))
        , T <= X;
    T-X, A :: total(A, T), prefer(le, (all, A, X), (P, W)), T
        >= X;
    X-T, A :: total(A, T), prefer(ge, (all, A, X), (P, W)), T
        <= X;
    T-X, A, C :: assign(C, A, T), prefer(le, (C, A, X), (P, W)
        ), T >= X;
    X-T, A, C :: assign(C, A, T), prefer(ge, (C, A, X), (P, W)
        ), T <= X
} : prefer((ge;le), (_, _, _), (P, _)).


% Prefering or disprefering certain components
#preference(comp(P), less(weight))
{
    -W, C :: component(C), domain(C, A, V), prefer(comp, (C, A
        , V), (P, W));
    -W, C :: not component(C), domain(C, A, V), prefer(negcomp
        , (C, A, V), (P, W))
} : prefer((comp;negcomp), (_, _, _), (P, _)).

```

```
34
35 %Prefering or disprefering certain attributes (color, material
       , ...)
36 #preference(attr(P), less(weight))
37 {
38     -W, C, A, V :: assign(C, A, V) , domain(C, A, V), prefer(
           attr, (C, A, V), (P, W));
39     -W, C, A, V :: assign(C, A, V) , domain(C, A, V), prefer(
           attr, (all, A, V), (P, W));
40     -W, C, A, V :: not assign(C, A, V) , domain(C, A, V),
           prefer(negattr, (C, A, V), (P, W));
41     -W, C, A, V :: not assign(C, A, V) , domain(C, A, V),
           prefer(negattr, (all, A, V), (P, W))
42 } : prefer((attr;negattr), (_, _, _), (P, _)).
43
44
45 % Pareto preference to list the pareto front for the given
       preferences
46 #preference(par, pareto) {**boundry(X); **attr(X); **weight(X)
       ; **comp(X)}.
47 #optimize(par).
```

4.2.1 Diversity in asprin is not guaranteed if only a limited amount of optimal solutions is shown. While can include diversity preferences in asprin, due to missing priorities less similar solutions might be chosen or user preferences could be ignored in order to accommodate for the diversity preferences. If a sufficiently large number of solutions is gathered through asprin, diverse solutions could then be selected in a separate step.

5 Examples

To show how preferences can be used in practice, I will go over the configuration process of a fictional user step-by-step. All requirements by the user will be represented using preferences. I will show the solutions found via optimization statements first and then compare them to the solutions found via asprin. Diversity preferences are added after each solution found in the case of optimization statements. For each time that a component type is chosen, the weight of the corresponding diversity preference is increased by 1.

Since there is no clear ranking to the answers given via asprin I will check if all solutions implement all preferences and if the solutions found with optimizations statements are included in the solutions found by asprin. If this is the case, then it would be reasonable to assume that equally diverse solutions can be extracted from the set of solutions provided by asprin.

The general configuration and recommendation process is shown in Figure 4. The user first sets some initial requirements. Then the configurator starts outputting solutions and, in the case of optimization statements, diversity preferences are added after each solution given. A set of 3 solutions is then shown to the user. If the user wishes to continue the configuration process, they can tweak their requirements or select a solution they like and start looking for similar solutions. After adding the similarity preferences, the user is then also given the option to tweak their requirements.

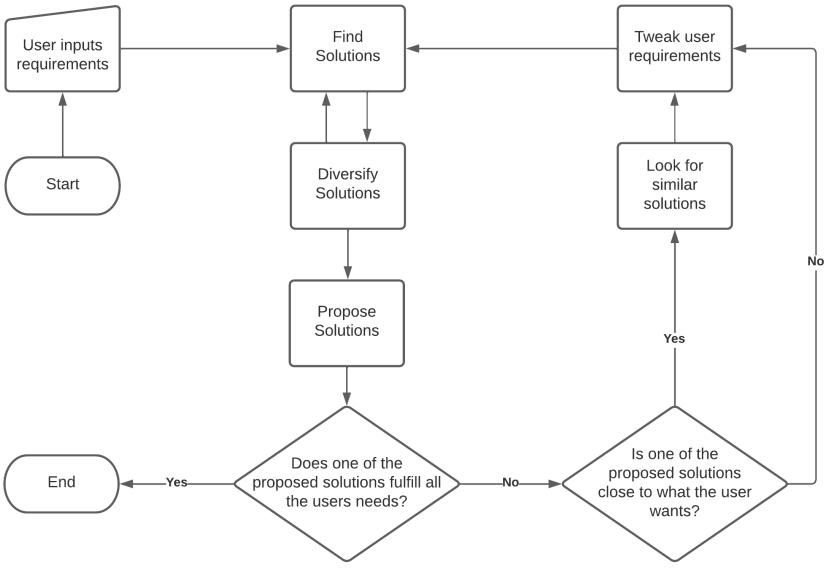

Fig. 4. A representation of the recommendation process used in the examples

5.1 Example 1

The user wants a bike with a basket. A basket also requires a stand. The user says that they do not want any other optional components and sets their price limit to 400. Since the user is clearly willing to pay a certain amount, the seller adds an additional preference to keep the price above 360, if possible, in order to increase profits.

The encoding of the preferences is given in Listing 3. A visual representation is given in Figure 5. The user requests have the highest priority. After that, the upper price limit is introduced in line 7. Line 8 describes the lower price limit added by the seller at the lowest priority.

After an optimal solution has been found, diversity preferences are added to the encoding. With a priority of 5 diversity is only selected for among the solutions which best fulfill all other preferences. The diversity preferences added after each solution are given in Listing 4 and Listing 5. The first 3 solutions found this way are then recommended.

The solutions found with optimization statements are given in Figure 6.

The effect of diversity preferences can be clearly seen in this first example. For example, each recommended solution uses a different basket. When the user selects a solution to use as a starting point for further steps, it is reasonable to assume that the basket used in the selected solution fits the customers preferences, even though they did not explicitly state a preference for that particular basket.

Using asprin a total of 5403 optimal solutions were found. The large number of solutions is due to the LE and GE preferences since every possible solution with a basket, a stand, no other optional components, and a price between 360 and 400 is optimal. The 3 solutions found by the optimization statements are included. This can lead us to conclude that diversity within these answers is provided, however it is not guaranteed that the first n solutions given are diverse. All preferences are satisfied in every solution. Further processing would be needed to filter and rank the solutions.

5.2 Example 2

The user now continues the configuration process. They like the first solution that was proposed and want to see similar solutions. For this example it is assumed the material of the frame as well as the size and material of the wheels are the most defining attributes of a bike and create similarity preferences for them. Additionally, since the user specifically requested a basket (and a stand by extension) it is also reasonable to include similarity preferences for the stand height and the material of the basket as these seem relevant to the user. The price preferences are also changed so that any proposed solution is within 10% of the price of the reference solution. The priority of the GE preference is increased, as it is now part of the similarity-finding process. The encoding is given in Listing 6. The newly added similarity preferences can be found in line 11-17. A visual representation is given in Figure 7.

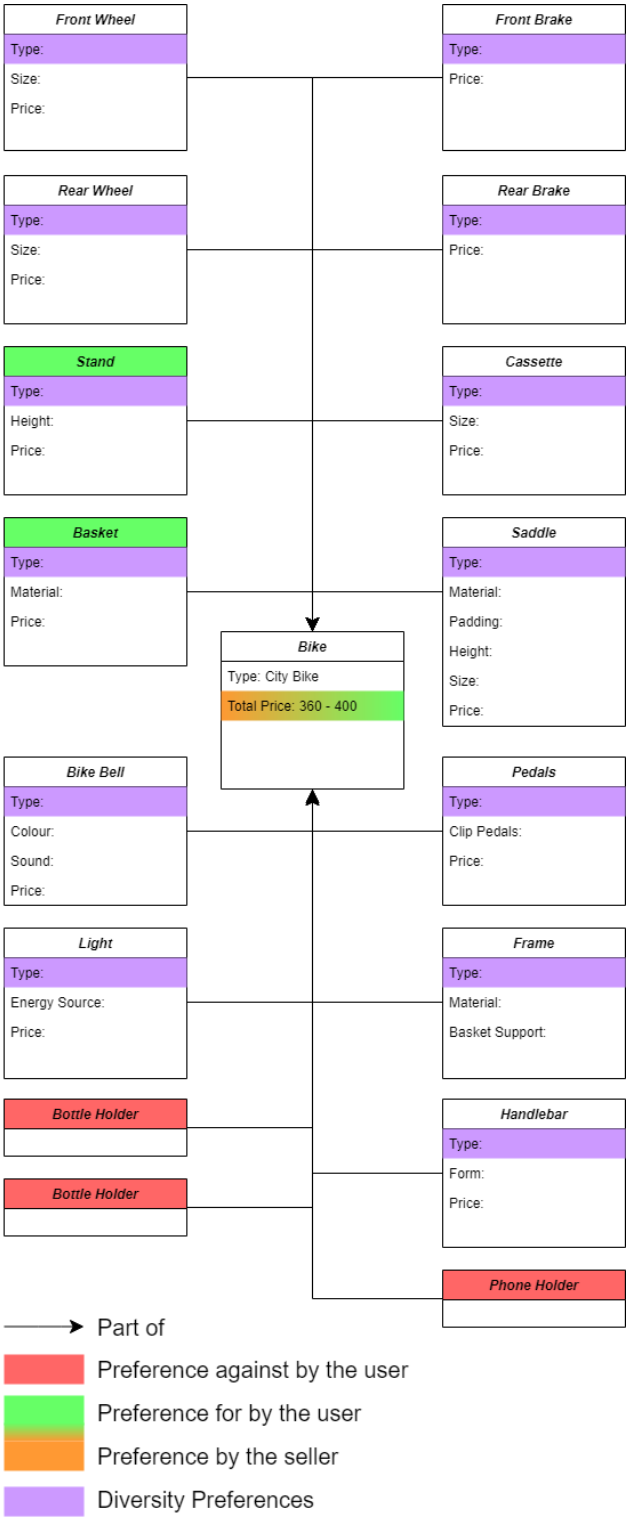

Fig. 5. A representation of the preferences used in Example 1

```
First solution:
Bike: City Bike
Saddle: s1
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb2
Light: l1
Cassette: c1
Pedals: p1
Front Brake: fb1
Rear Brake: rb2
Bell: bb1
Total Price: 370
```

```
Second solution:
Bike: City Bike
Saddle: s2
Stand: st2
Basket: b2
Rear Wheel: w1
Front Wheel: w3
Frame: f1
Handlebar: hb3
Light: l2
Cassette: c2
Pedals: p2
Front Brake: fb2
Rear Brake: rb1
Bell: bb2
Total Price: 382
```

```
Third solution:
Bike: City Bike
Saddle: s2
Stand: st2
Basket: b3
Rear Wheel: w3
Front Wheel: w1
Frame: f1
Handlebar: hb1
Light: l1
Cassette: c2
Pedals: p1
Front Brake: fb2
Rear Brake: rb2
Bell: bb2
Total Price: 371
```

Fig. 6. The first 3 solutions found with optimization statements and diversity preferences for Example 1

The solutions found with optimization statements are given in Figure 8.

Diversity preferences are still being added after each solution but will no longer be listed explicitly here.

Asprin gives 158 optimal solutions for the 2nd example. All 3 solutions found with optimization statements are among the 158 found by asprin this time as well.

5.3 Example 3

Again, the user likes the first solution. They decide to check options for a phone holder. At the same time, now that everything is the way the user wants it to, they want to check for a cheaper solution and check the "lowest price" option. The encoding is given in Listing 7. A visual representation is given in Figure 9.

The preference for a phone holder is added at the highest priority since it is the newest request by the user. This means the preference against a phone holder will be ignored, because a higher priority preference directly contradicts it.

The preference for the lowest price is performed last with the lowest priority as it is very restrictive and will often reduce the number of available options for following preferences severely. Additionally, the GE preference on the price is removed, as it would interfere with the minimization of the price.

The solutions found with optimization statements are given in Figure 10.

While the first solution is the cheapest possible solution, the other 2 are affected by the diversity preferences. Increasing the priority of the Min preference to be higher than the diversity preferences effectively disables the diversity preferences as Min reduces the pool of optimal answers to 1. Increasing the priority

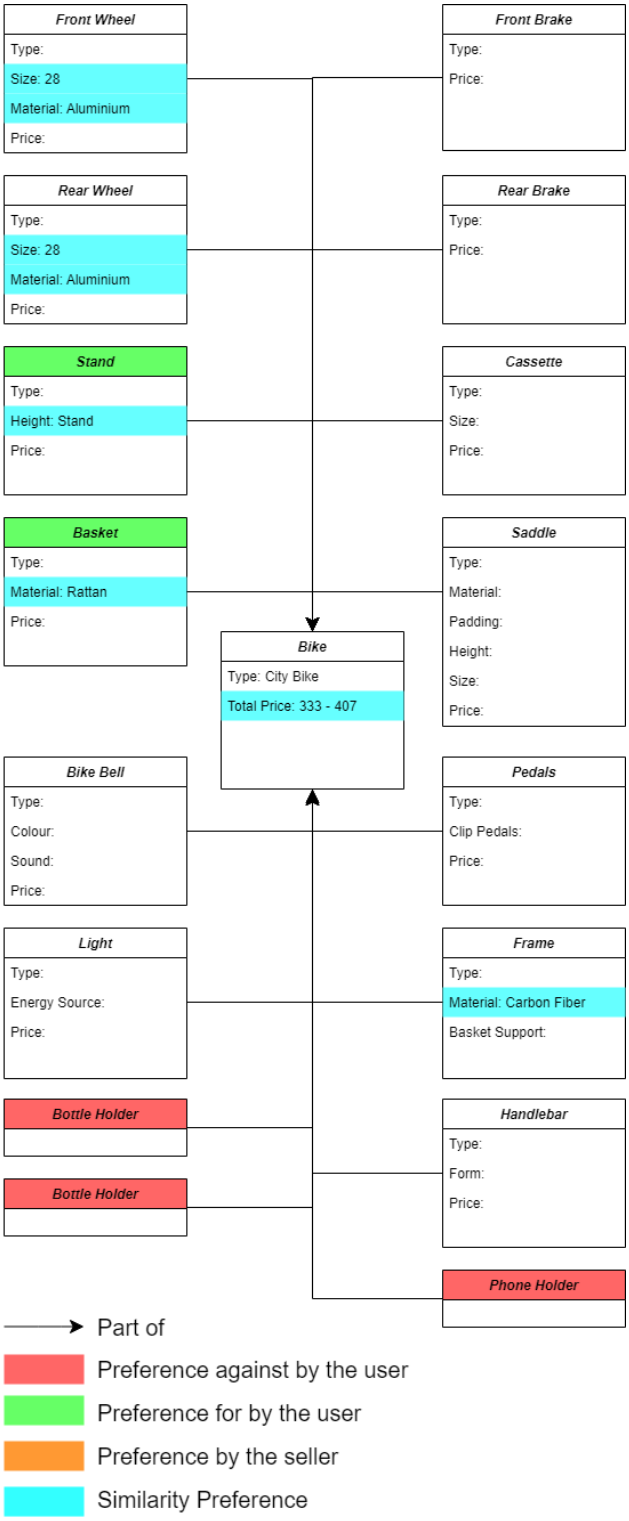

Fig. 7. A representation of the preferences used in Example 2

```
First solution:
Bike: City Bike
Saddle: s2
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb1
Light: l2
Cassette: c1
Pedals: p2
Front Brake: fb2
Rear Brake: rb2
Bell: bb2
Total Price: 339
```

```
Second solution:
Bike: City Bike
Saddle: s1
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb2
Light: l1
Cassette: c1
Pedals: p1
Front Brake: fb1
Rear Brake: rb1
Bell: bb1
Total Price: 394
```

```
Third solution:
Bike: City Bike
Saddle: s1
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb3
Light: l1
Cassette: c1
Pedals: p1
Front Brake: fb1
Rear Brake: rb2
Bell: bb2
Total Price: 342
```

Fig. 8. The first 3 solutions found with optimization statements and diversity preferences for Example 2

of the Min preference results in solutions that only differ in 1 component from the first solution found. Lowering the target value of the LE preference would improve this situation, but an idea as to what the cheapest possible solution is is needed first before adjusting the LE preference.

Asprin gives 45 optimal solutions. Only the first solution found via optimization statements is included. Some of the solutions do not include a basket. This is due to the Min preference. Since the inclusion of a basket increases the price, some solutions will forego the basket in favor of a cheaper solution. Min and Max type preferences may need to be handled separately from the pareto optimization in the future.

5.4 Evaluation

The priority-driven optimization statements can quickly produce a small set of diverse solutions. On the other handside, asprin can give a large amount of solutions, but, unless all answers are considered, diversity is not guaranteed and no ranking of solutions is given.

When it comes to minimizing a value, both approaches include the optimal solution that would be expected given the context but any additional solutions given by both implementations are of little use. In the case of optimization statements the minimization either prevents diversity or is prevented by diversity. In asprin the minimization preference can cause other preferences to be ignored in favor of minimizing the value even further. Using a high priority LE preference after the first optimal solution has been found would be a way to resolve this problem.

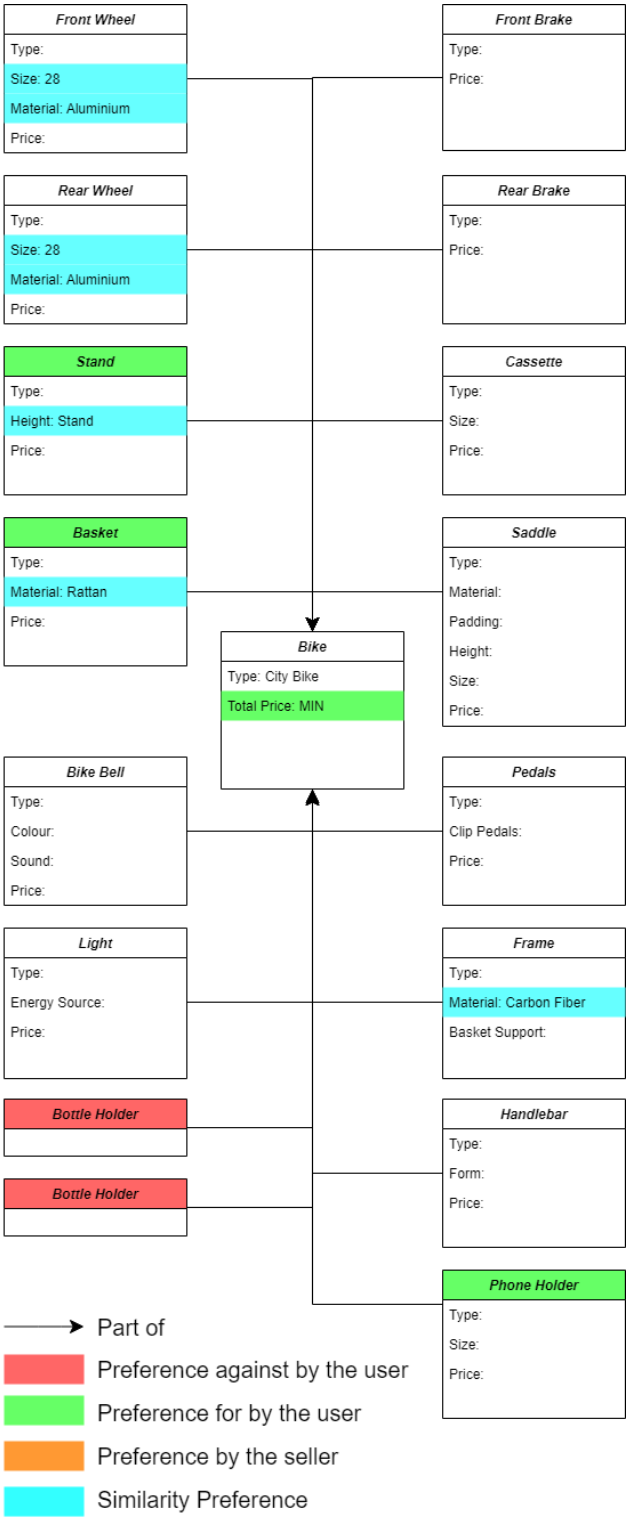

Fig. 9. A representation of the preferences used in Example 3

```
First solution:
Bike: City Bike
Saddle: s2
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb3
Light: l1
Cassette: c1
Pedals: p1
Front Brake: fb1
Rear Brake: rb2
Bell: bb2
Phone Holder: ph1
Total Price: 327
```

```
Second solution:
Bike: City Bike
Saddle: s2
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb1
Light: l2
Cassette: c1
Pedals: p2
Front Brake: fb2
Rear Brake: rb2
Bell: bb1
Phone Holder: ph1
Total Price: 362
```

```
Third solution:
Bike: City Bike
Saddle: s2
Stand: st1
Basket: b1
Rear Wheel: w2
Front Wheel: w2
Frame: f3
Handlebar: hb3
Light: l1
Cassette: c1
Pedals: p1
Front Brake: fb1
Rear Brake: rb1
Bell: bb2
Phone Holder: ph2
Total Price: 366
```

Fig. 10. The first 3 solutions found with optimization statements and diversity preferences for Example 3

Overall, the proposed systems are shown to be able to represent different types of user requests as well as similarity, diversity and owner requests. The exact results depend a lot on how exactly the preferences are being used.

6 Conclusion

This thesis expands the general purpose product configurator introduced in [4] by preferences, creating the basis for a similarly general purpose recommender. The recommender is then built with ASP and asprin, an extension for clingo. The FindMe systems are used as an inspiration.

Different preference types are introduced to encode possible requests of users. Priority and weight are also added as means to better control the recommendations. It is then shown how, on top of simple user requests, the concepts of similarity and diversity can be enforced.

These preferences are then encoded, once with clingo's built-in optimization statements and once with asprin's preferences. Using optimization statements, similarity, diversity and user preferences are expressed. Every preference requires a specific priority, but a user may have priorities different from the norm.

The resulting encoding works very similar to FindMe systems [3].

Using asprin, only similarity preferences are expressed in the encoding. However, the easy implementation of the pareto front allows for recommendations that do not require knowledge about priorities.

The usage of the encodings is then shown in a few example cases, showing examples of how to perform similarity-finding, diversification and tweaking in practice.

7 Future Work

While independence from priorities in the case of asprin presents advantages and disadvantages, a combination of both approaches implementing different priority levels of pareto preferences may show promise. Some preferences have been selected manually in the example. An intelligent recommender system that sets and adjusts preferences during the configuration process would be the next step. Priorities are currently fixed based on the component. Automatically adjusting these priorities according to the users needs and choices during the configuration process is another possible topic for future research. For this purpose, a system that can extract the user needs via a question and answer dialogue can prove useful.

Conflicts and contradictions are resolved either by different priorities or by providing different alternatives. However, the user will not be notified of such conflicts. Recognizing these conflicts and giving feedback on inconsistencies is an important part of product recommender systems.

The example given here shows that relevant solutions can be provided with the given preferences, but an in depth evaluation of the accuracy and precision will be required. The performance of this system on industry-scale knowledge-bases is also unknown.

8 Zusammenfassung

In dieser Arbeit wird das Konfigurationssystem aus [4] um ein wissensbasiertes Empfehlungssystem erweitert. Das Empfehlungssystem ist basierend auf Präferenzen aufgebaut.

Das Konzept von Präferenzen wird eingeführt und erklärt. Basierend darauf wird eine Liste von verschiedenen Präferenztypen erläutert und erarbeitet, die ein breites Spektrum an möglichen Nutzeranfragen abdecken können. Darüber hinaus werden weitere Konzepte wie Ähnlichkeit, Diversität und Standardwerte erklärt.

Diese Konzepte werden dann auf zweierlei Art in ASP kodiert. Einmal mit Hilfe der in clingo eingebauten *weak-constraints* und einmal mit Hilfe der clingo-Erweiterung asprin und der darin enthaltenden Paretopräferenz. Die Umsetzung der genannten Konzepte von Ähnlichkeit, Diversität und Standardwerten wird diskutiert und umgesetzt.

Es wird die Nutzung der Präferenzen an einem Beispielfall gezeigt und die Ergebnisse der beiden Umsetzungen werden verglichen. Obwohl die Konzepte von Diversität und Standardwerten in Asprin nur begrenzt umsetzbar sind, wird durch eine große Breite an Lösungen besonders Diversität sichergestellt. Die hohe Anzahl an optimalen Lösungen in Asprin bietet sich besonders für die Zusammenarbeit mit hybriden Empfehlungssystemen an.

Zukünftige Arbeiten können sich auf die Erkennung und Erläuterung von Konflikten in den Nutzeranforderungen, ein System zur Erkennung und Erarbeitung von Nutzeranforderungen und Prioritäten oder eine genaue Evaluation des Empfehlungssystems bezüglich der Genauigkeit und Präzision vornehmen.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering **17**(6), 734–749 (2005). https://doi.org/10.1109/TKDE.2005.99
2. Burke, R.: Knowledge-based recommender systems. Encyclopedia of library and information systems **69**(Supplement 32), 175–186 (2000)
3. Burke, R.D., Hammond, K.J., Yound, B.: The findme approach to assisted browsing. IEEE Expert **12**(4), 32–40 (1997)
4. Böke, C.O.: Knowledge-based design of a customizable product configurator (2021)
5. Falkner, A., Felfernig, A., Haag, A.: Recommendation technologies for configurable products. Ai Magazine **32**(3), 99–108 (2011)
6. Felfernig, A., Burke, R.: Constraint-based recommender systems: technologies and research issues. In: Proceedings of the 10th international conference on Electronic commerce. pp. 1–10 (2008)
7. Felfernig, A., Falkner, A., Müslüm, A., Erdeniz, S.P., Uran, C., Azzoni, P.: Asp-based knowledge representations for iot configuration scenarios. In: 19th International Configuration Workshop. vol. 62 (2017)
8. Felfernig, A., Hotz, L., Bagley, C., Tiihonen, J.: Knowledge-based configuration: From research to business cases. Newnes (2014)
9. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., Wanko, P.: The potassco user guide (2021)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Logic Programming **2** (12 2000)
11. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An asp-based system for e-tourism. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 368–381. Springer (2009)
12. Kaminski, R., Romero, J., Schaub, T., Wanko, P.: How to build your own asp-based system?! CoRR **abs/2008.06692** (2020), `https://arxiv.org/abs/2008.06692`
13. Mishra, S.: Product configuration in answer set programming. Electronic Proceedings in Theoretical Computer Science **345**, 296–304 (sep 2021). https://doi.org/10.4204/eptcs.345.46, `https://doi.org/10.4204%2Feptcs.345.46`
14. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: International Symposium on Practical Aspects of Declarative Languages. pp. 305–319. Springer (1999)
15. Tiihonen, J., Felfernig, A.: Towards recommending configurable offerings. International Journal of Mass Customisation **3**(4), 389–406 (2010)
16. Zhang, L.: Product configuration: A review of the state-of-the-art and future research. International Journal of Production Research **52**, 6381–6398 (05 2014). https://doi.org/10.1080/00207543.2014.942012

A Encodings

In the previous sections different encodings have been mentioned. The full encodings are given in this section.

Listing 3. The encoding of the first test case

```
1  prefer(comp, (basket, 0, 0), (110, 20)).
2  prefer(comp, (stand, 0, 0), (110, 20)).
3  prefer(negcomp, (phone_holder, 0, 0), (110, 20)).
4  prefer(negcomp, (bottle_holder, 0, 0), (110, 20)).
5  prefer(negcomp, (seat_cover, 0, 0), (110, 20)).
6
7  prefer(le, (all, price, 400), (90, 20)).
8  prefer(ge, (all, price, 360), (5,20)).
```

Listing 4. The encoding of diversity preferences in the first test case after the first solution found

```
1  prefer(negattr, (bike,type,city_bike),(5, 1)).
2  prefer(negattr, (saddle,type,s1),(5, 1)).
3  prefer(negattr, (stand,type,st1),(5, 1)).
4  prefer(negattr, (basket,type,b1),(5, 1)).
5  prefer(negattr, (rear_wheel,type,w2),(5, 1)).
6  prefer(negattr, (front_wheel,type,w2),(5, 1)).
7  prefer(negattr, (frame,type,f3),(5, 1)).
8  prefer(negattr, (handlebar,type,hb2),(5, 1)).
9  prefer(negattr, (light,type,l1),(5, 1)).
10 prefer(negattr, (cassette,type,c1),(5, 1)).
11 prefer(negattr, (pedals,type,p1),(5, 1)).
12 prefer(negattr, (front_brake,type,fb1),(5, 1)).
13 prefer(negattr, (rear_brake,type,rb2),(5, 1)).
14 prefer(negattr, (bell,type,bb1),(5, 1)).
```

Listing 5. The encoding of diversity preferences in the first test case after the second solution found

```
1  prefer(negattr, (bike,type,city_bike),(5, 2)).
2  prefer(negattr, (saddle,type,s1),(5, 1)).
3  prefer(negattr, (stand,type,st1),(5, 1)).
4  prefer(negattr, (basket,type,b1),(5, 1)).
5  prefer(negattr, (rear_wheel,type,w2),(5, 1)).
6  prefer(negattr, (front_wheel,type,w2),(5, 1)).
7  prefer(negattr, (frame,type,f3),(5, 1)).
8  prefer(negattr, (handlebar,type,hb2),(5, 1)).
9  prefer(negattr, (light,type,l1),(5, 1)).
10 prefer(negattr, (cassette,type,c1),(5, 1)).
11 prefer(negattr, (pedals,type,p1),(5, 1)).
12 prefer(negattr, (front_brake,type,fb1),(5, 1)).
13 prefer(negattr, (rear_brake,type,rb2),(5, 1)).
14 prefer(negattr, (bell,type,bb1),(5, 1)).
```

```
prefer(negattr, (bike,type,city_bike),(5, 2)).
prefer(negattr, (saddle,type,s2),(5, 1)).
prefer(negattr, (stand,type,st2),(5, 1)).
prefer(negattr, (basket,type,b2),(5, 1)).
prefer(negattr, (rear_wheel,type,w1),(5, 1)).
prefer(negattr, (front_wheel,type,w3),(5, 1)).
prefer(negattr, (frame,type,f1),(5, 1)).
prefer(negattr, (handlebar,type,hb3),(5, 1)).
prefer(negattr, (light,type,l2),(5, 1)).
prefer(negattr, (cassette,type,c2),(5, 1)).
prefer(negattr, (pedals,type,p2),(5, 1)).
prefer(negattr, (front_brake,type,fb2),(5, 1)).
prefer(negattr, (rear_brake,type,rb1),(5, 1)).
prefer(negattr, (bell,type,bb2),(5, 1)).
```

Listing 6. The encoding of the second test case

```
prefer(comp, (basket, 0, 0), (110, 20)).
prefer(comp, (stand, 0, 0), (110, 20)).
prefer(negcomp, (phone_holder, 0, 0), (110, 20)).
prefer(negcomp, (bottle_holder, 0, 0), (110, 20)).
prefer(negcomp, (seat_cover, 0, 0), (110, 20)).

prefer(le, (all, price, 407), (90, 20)).
prefer(ge, (all, price, 333), (90, 20)).

% Similarity preferences
prefer(attr, (frame,material,carbon_fiber),(50, 20)).
prefer(attr, (front_wheel,material,aluminium),(60, 20)).
prefer(attr, (front_wheel,size,26),(60, 20)).
prefer(attr, (rear_wheel,material,aluminium),(60, 20)).
prefer(attr, (rear_wheel,size,26),(60, 20)).
prefer(attr, (basket,material,rattan),(30, 20)).
prefer(attr, (stand,height,15),(30, 20)).
```

Listing 7. The encoding of the third test case

```
prefer(comp, (phone_holder, 0, 0), (120,20)).

prefer(comp, (basket, 0, 0), (110, 20)).
prefer(comp, (stand, 0, 0), (110, 20)).
prefer(negcomp, (phone_holder, 0, 0), (110, 20)).
prefer(negcomp, (bottle_holder, 0, 0), (110, 20)).
prefer(negcomp, (seat_cover, 0, 0), (110, 20)).

prefer(le, (all, price, 372), (90, 20)).


prefer(attr, (frame,material,carbon_fiber),(50, 20)).
prefer(attr, (front_wheel,material,aluminium),(60, 20)).
```

```
14 prefer(attr, (front_wheel,size,26),(60, 20)).
15 prefer(attr, (rear_wheel,material,aluminium),(60, 20)).
16 prefer(attr, (rear_wheel,size,26),(60, 20)).
17 prefer(attr, (basket,material,rattan),(30, 20)).
18 prefer(attr, (stand,height,15),(30, 20)).
19
20 prefer(min, (all, price, 0), (1, 1)).
```