COMPSCI 767 Case-Based Reasoning Report

Helen Zhao

May 28, 2017

Contents

1 Introduction

Case-cased reasoning (CBR) is a methodology of problem solving, which uses solutions of past problems to solve a new problem. This report will describe the use of case-based reasoning in a travel-suggestion application. The user can input requirements for the trip, and the system will retrieve the most similar case(s) to advise the user.

Section 2 details the design process, and section 3 contains instructions for installation.

2 Design

2.1 Overview

The program contains 1024 previous travel cases as the database, which is imported as a .csv file at the start of the program. Each case is stored in a `Case` class, which stores all the necessary attributes for the cases. Each case contains 10 attributes: JourneyCode, HolidayType, Price, NumberOfPersons, Region, Transportation, Duration, Season, Accommoation, and Hotel. Out of the attributes, only 9 will be considered for similarity, since JourneyCode is an identifier and does not provide meaningful information for the content of the case. The weights for each attribute are initiated at the beginning of the program.

The program will ask a question for each of the 9 attributes, at any point the user can press 'Enter' to skip the attribute. Each question will also specify the types of input they are expecting, whether it's a numeric value or a set of strings as possible inputs. At the end, the user will also be asked regarding how many results they would like to see. Finally the program will perform retrieval and return the k most similar cases to the case which the user desires, with k being the number of top results specified by the user. After displaying the results, the user will be given the choice of entering a new query or to end the program.

Relevant Functions

- Main function:
  `void main(String[] args)`
- Construct cases from .csv:
  `void constructCases()`
- Construct weights:
  `void constructWeights()`
- Obtain user input:
  `Case obtainCase()`

- Retrieve similarities for all cases:
  `ArrayList<Entry<Double,Case>> retrieveCases (Case userCase)`
- Return top results to user:
  `void returnResults(ArrayList<Entry<Double,Case>> topCases)`

2.2 Retrieval

The similarity between two cases can be represented using two aspects:

1. **Local similarity** for each attribute which is calculated individually.
2. **Relative relevance** (weights) of each attribute.

Mathematically, this can be represented as the following:

$$sim(x, y) = \sum_{i=1}^{i=n} w_i sim(x_i, y_i)$$

The similarity between the case representing the user's input, and every 1024 cases in the database is calculated. The similarity between each case is then entered into a `TreeMap` data structure, which stores the elements in a sorted ascending order based on a key (similarity) into a tree. Finally, once the similarity with all cases has been calculated and entered into the `TreeMap`, the program returns k results, consecutively removed from `TreeMap` which ensures that they are the cases consisting of the highest similarity values.

Relevant Functions

- Retrieve similarities for all cases:
  `ArrayList<Entry<Double, Case>> retrieveCases (Case userCase)`
- Calculate similarity between two cases:
  `double calculateSimilarity(Case c, Case userCase)`
- Return top results to user:
  `void returnResults(ArrayList<Entry<Double, Case>> topCases)`

2.3 Local Similarity Metrics

This section will describe the local similarity function between each of the 9 attributes. For the numeric entries, the similarity is calculated using the following formula:

$$diff(n_{case}, n_{input}) = \frac{|n_{case} - n_{input}|}{max(n_{case}, n_{input})} \quad (1)$$

$$sim(n_{case}, n_{input}) = 1 - diff(n_{case}, n_{input}) \quad (2)$$

The first equation calculates the distance between the two values, normalised to a value between 0 and 1. The second equation finds the similarity using the difference, which also returns a value between 0 and 1.

2.3.1 Holiday Type

In the case database, holiday types can consist one of the following: city, education, language, recreation, bathing, wandering, active, and skiing. A taxonomy tree was used to construct the similarity function for this attribute:

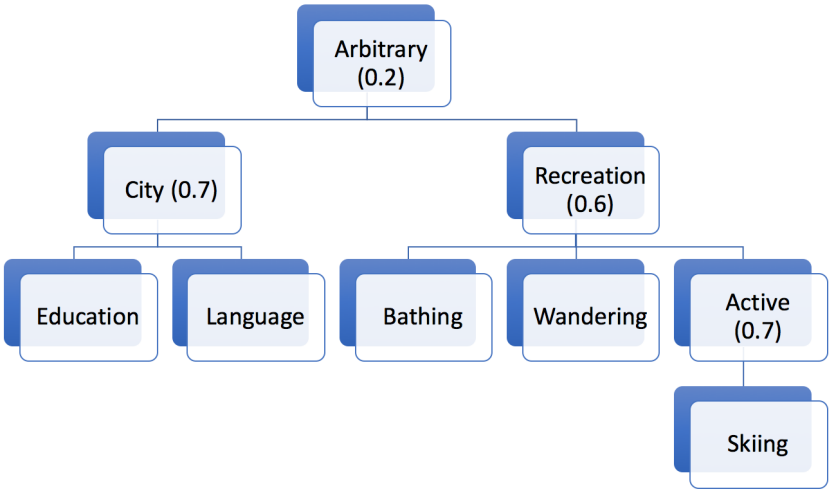

Figure 1: Taxonomy tree for HolidayType attribute

Since the input is textual, automatic spelling-correction is provided. The input will be checked against one of the valid types. If it is a valid input then the similarity will be found using the taxonomy tree, and if not, the Levenshtein distance between the input and every valid type will be calculated, and the input is replaced with the most similar valid type.

The similarity between any two holiday types can be found by finding the least common ancestor and using the similarity value for that node. As an example, the similarity between "Bathing" and "Wandering" is 0.6 since their lowest common ancestor is "Recreation (0.6)", and the similarity between "Bathing" and "City" is 0.2 since their lowest common ancestor is "Arbitrary (0.2)". In the program, the taxonomy tree is stored as a 2D matrix for ease of retrieval, with the two indices of the matrix each representing a holiday type. In the implementation, the data structure uses a decision table for ease of retrieval.

The holiday type attribute is given a weight of 3 since the types are quite generic, and people sometimes want very specific types of holidays

Relevant Functions

- Find holiday type similarity between two cases:
  `double holidayTypeSimilarity(Case c, Case userCase)`

- Construct a hashmap for every type and its numerical code:
  `HashMap<String, Integer> constructTypeHashMap()`
- Construct a 2D matrix to use as a decision table:
  `double[][] constructTypeMatrix()`
- Find the most similar valid input:
  `String findMostSimilarInput(String s, HashSet<String> set)`

2.3.2 Price

The price of the user input is considered as a budget, which is the maximum amount of money the user is willing to spend. The price in user input is usually also dependant on the duration and number of people travelling. This comes from the intuition that a case with a price of 2000, party of 1 and duration of 1 day is a very different type of trip than a case with the same price, 2000, but a group of 5 people, and duration of 7 days. For this reason, we adjust the price from user input first before comparing for similarity. If the user has specified the number of people travelling and/or the duration of the trip, then the user budget will be averaged amongst those values to find the price for the trip per person per night.

After the trip price has been adjusted, the similarity between the prices are calculated using the numeric similarity function specified as above 2.

The price attribute is given a weight of 8 since financial reasons is one of the most important factors in a holiday, and therefore it has been given the highest weighting.

Relevant Functions

- Find price similarity between two cases:
  `double priceSimilarity(Case c, Case userCase)`

2.3.3 Number of Persons

Number of persons can be seen as the user wishes the trip can accommodate *at least* this number of people, since when people plan trips they have an idea of who they would like to go with and this may not very flexible. For this reason, we construct the similarity function such that a similarity of 0 will be given to every case where the number of persons is less than the input. For all cases with number of persons at least that of the input, the similarity will be calculated with the function 2 as specified above.

This attribute is given a weight of 6 since as previously stated, the number of people on the trip may not be very negotiable and therefore should be given a higher weighting.

Relevant Functions

- Find number of persons similarity between two cases:
  `double numPersonsSimilarity(Case c, Case userCase)`

2.3.4 Region

The region is an interesting case since a user may choose to travel in a particular region for various reasons. Therefore in order to calculate the similarity between two regions, we should take into consideration the possible reasons for the region choice. The observation here is that people generally choose a particular region because they prefer that particular country or surrounding areas, or they choose a region for reasons such that it contains mountains, coasts, or have historic relevance.

For this reason, we generalise the user intention into two categories and construct a taxonomy tree for each: choosing a particular region by the geological **location** or choosing a particular region by the land **features**. For each taxonomy tree, the regions are categorised such that the regions within the same subtree of the taxonomy tree have a greater similarity.

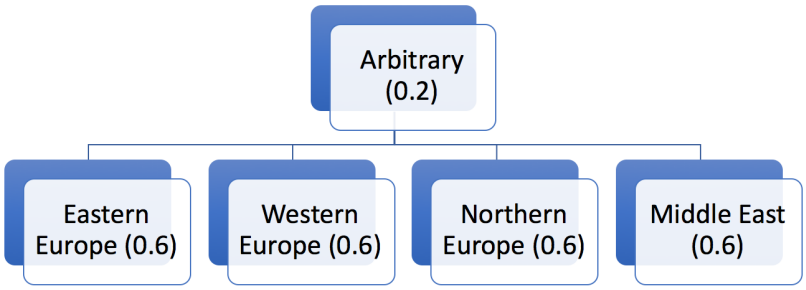

Figure 2: Taxonomy tree for choosing region by location

For reasons regarding space, the regions will not be listed but they are all direct children of one of the leaf nodes for each graph.

At the case acquisition phase, the user is presented with a decision-tree interaction where they can choose which mode they would like to select the region by, and following each mode, they will be presented choices of the categories which they can also choose, and all regions under that particular category will be listed so that the user could choose them as input. The similarity will be calculated with the relevant taxonomy-tree which the user has specified at the point of input. If the user has not made a decision between the location mode

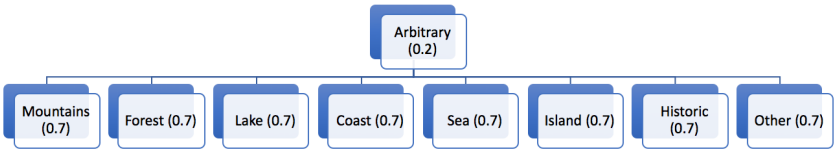

Figure 3: Taxonomy tree for choosing region by features

or the feature mode, they will be presented with all possible regions and the similarity will be calculated by the location taxonomy tree.

Just like the holiday type attribute, automatic spelling-correction is provided. The input will be checked against one of the valid types and the misspelled input is replaced with the most similar valid region.

The region is given a weight of 5 since it is a very important aspect of travelling, however due to the uncertainty it has not been given a very high weighting.

Relevant Functions

- Find region similarity between two cases:
  `double regionSimilarity(Case c, Case userCase)`
- The decision process which helps the user to choose a relevant mode and to find relevant regions:
  `String obtainRegion(Scanner s)`
- Constructing a set of regions for each category:
  `void constructRegions()`

2.3.5 Transportation

The transportation mode can either be train, car, plane, and coach. The similarity between the transportation modes can vary since the user may be limited by their means of transportation, and for that reason we use asymmetric decision table for the similarity values as follows:

	Train	Car	Plane	Coach
Train	1	0.3	0.5	0.8
Car	0.7	1	0.1	0.7
Plane	0.5	0.1	1	0.3
Coach	0.8	0.3	0.3	1

Table 1: Decision table for transportation mode

We will explain the similarity values for each mode of transportation:

- Train: if the user input is *train*, we give *car* a relatively low similarity of 0.3 since the user may not have access to a car. However *coach* will have a high similarity of 0.8 since as public transports they are quite interchangeable. *Plane* on the other hand, is assigned a similarity of 0.5 since price may be different.
- Car: *car* has a high similarity of 0.7 to both *train* and *coach*, since they are all land vehicles. However it has a low similarity of 0.1 to *plane*, since for any destination which can be reached in car, the user will be unlikely to want to take a plane.
- Plane: *plane* is quite a specialised mode of transport, and for this reason it has relatively low similarities for all other modes: 0.5 for *train*, 0.1 for *car* and 0.3 for *coach*. The reasoning being that it perhaps can be reached by train since they are both long-distance vehicles, however car and coach have lower similarity since they are usually used for shorter distances.
- Coach: This mode of transport has 0.8 similarity to train since they are quite similar, 0.3 for *car* for the same reasoning as stated train, and 0.1 for plane which is lower similarity than train since train is more catered to long-distance than coach.

Similar to previously explained text-based attributes, the same spell-correction will be carried out and replaced with the most similar valid region.

The region is given a weight of 3 since people could be quite flexible with their mode of transport.

Relevant Functions

- Find transportation similarity between two cases:
  `double transportationSimilarity(Case c, Case userCase)`
- Constuct a hashmap for every type and its numerical code:
  `HashMap<String, Integer> constructTransports()`
- Construct a 2D matrix as decision table:
  `double[][] constructTransportMatrix()`

2.3.6 Duration

The duration is a numerical values, and is considered as an *upper limit* since most of the time people's travel time is constrained. People usually have a limited number of days they can take off work or studies, and we construct the similarity function with this in mind. We will assign a similarity of 0 for every case where the duration is longer than the user input, and the rest will be specified with the same function 2 as specified above.

The duration attribute is assigned a weight of 7 which is quite high, since the time constriction could be quite important to people and the similarity metrics is quite accurate.

Relevant Functions

- Find duration similarity between two cases:
  `double durationSimilarity(Case c, Case userCase)`

2.3.7 Season

The seasons attribute contains a value consisting of a month of the year, and to find the similarity between two months we have to take into consideration the season the month is in. Since all the locations are in the Northern Hemisphere, we consider seasons as categorised as the following:

- Spring: March - May
- Summer: June - August
- Autumn: September - November
- Winter: December - February

We consider the similarity between two months as follows: A similarity of 1 will be assigned to two months that are identical, and a similarity of 0.8 between two months that are 1 month apart.

Then, we will consider the similarity between the seasons the months are in. Firstly, two months are assigned a similarity of 0.8 if they are within the same season. Secondly, since spring and autumn generally have similar temperature and weather, they are assigned a similarity of 0.7. Thirdly, if two seasons are neighbouring seasons then they are assigned a similarity of 0.5. Finally, since summer and winter are complete opposite seasons, they are assigned a similarity of 0. This can be represented with the following decision table.

	Spring	Summer	Autumn	Winter
Spring	0.8	0.5	0.7	0.4
Summer	0.5	0.8	0.5	0
Autumn	0.7	0.5	0.8	0.5
Winter	0.5	0	0.5	0.8

Table 2: Decision table for seasons

The season attribute is assigned a weight of 4, since different seasons can make a big difference to the holiday, but the month is still an estimate and not necessarily the most accurate. For this reason it has not been assigned a very high weighting.

Relevant Functions

- Find season similarity between two cases:
  `double seasonSimilarity(Case c, Case userCase)`

2.3.8 Accommodation

The accommodation attribute can take one of the following options: HolidayFlat, OneStar, TwoStar, ThreeStars, FourStars, FiveStars. Holiday flat is inherently a very different type of accommodation to a hotel, it is a standalone house which the vacationer can rent and use as their own for their holidays. There are no hotel services, and it also offers a larger amount of space and freedom. For this reason, people who search for holiday flats compared to people who search for hotels want very different types of accommodations and sometimes those needs can be very specific.

We build our similarity function using this intuition, and holiday flats have a similarity of 1.0 to itself, and a similarity of 0 to everything else.

As for the star ratings, we interpret it as the user wants *at least* that many stars for their accommodation. Therefore we assign a similarity of 0 for all accommodations with the star number less than that is specified. Then we assign a similarity of 0.8 if the difference in stars is 1, 0.6 if 2, 0.4 if 3, and 0.2 if 4.

The accommodation attribute is assigned a weight of 4 since albeit the importance, but the star system is quite vague and usually it is not a priority for people when they are choosing accommodation, and it is a category that people sometimes are willing to compromise in.

Relevant Functions

- Find accommodation similarity between two cases:
  `double accommodationSimilarity(Case c, Case userCase)`
- Convert the accommodation to numeric stars for calculation:
  `int convertToNumericStars(String accom)`

2.3.9 Hotel

The hotel section consists of a list of strings which is the name of the hotel. To compare how similar the two hotel names are, we use Levenshtein distance but modified for a series of words. The distance represent number of operations needed to change one hotel name to the other, with each operation changing one word (instead of a letter).

To help the user with choosing an accommodation, a list of hotels which are from the region the user has specified are printed. If the user has not specified a region, then all hotels will be printed.

The accommodation attribute is assigned a weight of 2, since it is usually not the most important factor when planning location, and people usually are willing to change accommodation if it turns out that a particular hotel is full.

Relevant Functions

- Print relevant hotels to the user specified region:
  `void printRelevantHotels(String region)`

- Find hotel name similarity between two cases:
  `double hotelSimilarity(Case c, Case userCase)`
- Calculate distance between two strings of words:
  `int getSentenceDistance(String s1, String s2)`
- Calculate similarity between two strings of words:
  `double getSentenceSimilarity(String s1, String s2)`

3 Installation

The program is written in Java, with a command-line interface. There are two .java files: Case.java and Main.java. Case.java is the class structure for each travel case, and Main.java is the main program.

In order to run the executable travel.jar, place the file travel.csv in the same folder. Open a command line program at the location and enter the following command to execute the .jar file: `java -jar travel.jar`.