CmpE 260 - Principles of Programming Languages
Spring 2021 - Project 1

Deadline: 24 May 2021 09:00

Assistants: Alper Ahmetoğlu, Onur Sefa Özçıbık
ahmetoglu.alper@gmail.com, onursefa_ozcibik@hotmail.com

1 Introduction

In this project, you will implement a matching application in Prolog.

2 Knowledge Base

You have 7 main files each containing a single type of relation. These files constitute your knowledge base.

- `glanian(GlanianName, GlanianGender, GlanianFeatures)`

  Basic properties of a glanian.

  - `GlanianName`: name of the glanian.
  - `GlanianGender`: gender of the glanian.
  - `GlanianFeatures`: list of 10 main glanian features normalized in the range [0, 1]: *Age, Salary, Intelligence, Romance, Creativity, Responsibility, Fun, Kindness, Honesty, Good Looking.*

- `expects(GlanianName, ExpectedGenders, ExpectedFeatures)`

  Expectations of a glanian from her partner.

  - `GlanianName`: name of the glanian.
  - `ExpectedGenders`: list of genders which this glanian expects.
  - `ExpectedFeatures`: list of features which this glanian expects.

- `weight(GlanianName, WeightList)`

  Relative importance of the each feature for this glanian.

  - `GlanianName`: name of the glanian.
  - `WeightList`: list of weights containing real numbers in the range [0, 1].

- `likes(GlanianName, LikedActivities, LikedCities)`

  Activities and cities which this glanian likes.

  - `GlanianName`: name of the glanian.
  - `LikedActivities`: list of activities which this glanian likes.
  - `LikedCities`: list of cities which this glanian likes.

- `dislikes(GlanianName, DislikedActivities, DislikedCities, Limits)`

  Activities and cities which this glanian dislikes and her tolerable feature limits.

  - `GlanianName`: name of the glanian.
  - `DislikedActivities`: list of activities which this glanian dislikes.
  - `DislikedCities`: list of cities which this glanian dislikes.
  - `Limits`: list of lists containing tolerable limits for each feature. Each list contains two values: minimum and maximum limit. If the glanian does not have a specific limit, then the list is empty.

- `city(CityName, HabitantList, ActivityList)`

  Basic properties of a city.

  - `CityName`: name of the city.
  - `HabitantList`: list of habitants that live in this city.
  - `ActivityList`: list of activities that can be done in this city.

- `old_relation([GlanianName1, GlanianName2])`

  Old relationships.

  - GlanianName1: name of the first glanian.
  - GlanianName2: name of the second glanian.

3 Predicates

In this section, we will go over the predicates that you are going to implement.

3.1 `glanian_distance(+Name1, +Name2, -Distance)` 5 points

Given two glanians `Name1` and `Name2`, this predicate will return the distance from `Name1` to `Name2`. This distance is the Euclidean distance between `Name1`'s `ExpectedFeatures` and `Name2`'s `GlanianFeatures`. It is a measure of how closer `Name2` to `Name1`'s expectations. Notice that this predicate is not symmetric.

Let $e = [e_1, e_2, \ldots, e_{10}]$ be the `ExpectedFeatures` of `Name1` and $f = [f_1, f_2, \ldots, f_{10}]$ be the `GlanianFeatures` of `Name2`. Then, the glanian distance is calculated as follows:

$$D_{glanian} = \left(\sum_{e_i \neq -1} (e_i - f_i)^2 \right)^{1/2} \tag{1}$$

For example, if $e = [-1, 3, 4]$, $f = [2, 7, 8]$, then the glanian distance is $\sqrt{(3-7)^2 + (4-8)^2}$. We omit the first feature since $e_1 = -1$ (i.e. `Name1` does not care about the first feature).

Examples:

```
?- glanian_distance(zhuirlu, josizar, D).
   D = 1.218001642035018.
?- glanian_distance(josizar, zhuirlu, D).
   D = 0.8932983824008639.
?- glanian_distance(olisor, calemi, D).
   D = 1.0484364549175118.
?- glanian_distance(calemi, olisor, D).
   D = 1.2979672569059668.
```

3.2 `weighted_glanian_distance(+Name1, +Name2, -Distance)` 10 points

Given two glanians `Name1` and `Name2`, this predicate will return the weighted distance from `Name1` to `Name2`. It is almost the same as the first except that `Name1` puts weights to each features, showing her preferences.

Let $e = [e_1, e_2, \ldots, e_{10}]$ be the `ExpectedFeatures` of `Name1`, $w = [w_1, w_2, \ldots, w_{10}]$ be the `WeightList`, and $f = [f_1, f_2, \ldots, f_{10}]$ be the `GlanianFeatures` of `Name2`. Then, the weighted glanian distance is calculated as follows:

$$D_{glanian} = \left(\sum_{e_i \neq -1} w_i (e_i - f_i)^2 \right)^{1/2} \tag{2}$$

For example, if $e = [-1, 3, 4]$, $f = [2, 7, 8]$, $w = [-1, 0.5, 1]$, then the weighted glanian distance is $\sqrt{0.5(3-7)^2 + 1(4-8)^2}$.

Examples:

```
?- weighted_glanian_distance(zhuirlu, josizar, D).
   D = 0.7717511418844807.
?- weighted_glanian_distance(josizar, zhuirlu, D).
   D = 0.4353217993622649.
?- weighted_glanian_distance(olisor, calemi, D).
   D = 0.40758454337719924.
?- weighted_glanian_distance(calemi, olisor, D).
   D = 0.9851317196192598.
```

3.3 `find_possible_cities(+Name, -CityList)` 5 points

This predicate will return a list of cities which contains:

- The current city of `Name`.
- `Name`'s `LikedCities`.

Examples:

```
?- find_possible_cities(zhuirlu, CityList).
   CityList = [venis, beyroot, istenbol]
?- find_possible_cities(josizar, CityList).
   CityList = [corse_town, seviliri, viyan]
```

3.4 merge_possible_cities(+Name1, +Name2, -CityList) 5 points

Given two glanians `Name1` and `Name2`, this predicate will return the union of the two glanian's possible cities.

Examples:

```
?- merge_possible_cities(zhuirlu, josizar, CityList).
   CityList = [venis, beyroot, istenbol, corse_town, seviliri, viyan]
?- merge_possible_cities(zhuirlu, zhuirlu, CityList).
   CityList = [venis, beyroot, istenbol]
```

3.5 find_mutual_activities(+Name1, +Name2, -ActivityList), 5 points

Given two glanians `Name1` and `Name2`, this predicate will return the list of mutual activities of two glanians. An activity is mutual if it is in both glanians' `LikedActivities`.

Examples:

```
?- find_mutual_activities(zhuirlu, josizar, ActivityList).
   ActivityList = []
?- find_mutual_activities(zhuirzaz, josizar, ActivityList).
   ActivityList = [camping, swimming]
```

3.6 find_possible_targets(+Name, -Distances, -TargetList), 10 points

This predicate will return a list of possible glanians sorted by their distances as possible matching targets for `Name`. `Distances` should be a sorted list that contains `glanian_distance` from `Name` to glanians in `TargetList`. The gender of each glanian in `TargetList` should be in `ExpectedGenders` of `Name`. For example, if a glanian has an empty list in her `expects` entry, then the results should be an empty list since she does not expect any gender.

Examples:

```
?- find_possible_targets(zhuirlu, Distances, TargetList)
   Distances = [0.3006409819036653, 0.3238039530333131, 0.328,
   0.3493422390722312, 0.40817398251235953, 0.4136302696853797,
   0.4190071598433611, 0.42604577218885764, 0.427071422598141|...]
   TargetList = [golkolz, jai-blava, faeno, darcaluna, anfin, aidel, bloszen,]
   sheeanth, gallan|...]
?- find_possible_targets(zhuirzaz, Distances, TargetList)
```

```
Distances = [0.3532860031192857, 0.4758739328855911, 0.5260465758846834,
0.5502290432174586, 0.5630337467683442, 0.5718933467002392,
0.6119852939409575, 0.6136024771788328, 0.6186784302042538|...]
TargetList = [angwispm, engsangu, ranaqri, wistur, stermilky, faevine,
jodturv, wilkster, faezab|...].
```

3.7 find_weighted_targets(+Name, -Distances, -TargetList), 15 points

This predicate will return a list of possible glanians sorted by their *weighted* distances as possible matching targets for `Name`. It is the same as the previous predicate except that the distances are calculated with `weighted_glanian_distance(Name, Target, D)`.

Examples:

```
?- find_weighted_targets(zhuirlu, Distances, TargetList)
   Distances = [0.1385049818598595, 0.1692282511875603, 0.18459984019494705,
   0.2198129454786501, 0.2261861556329211, 0.24256776991183307,
   0.24317945842525432, 0.2508943801682293, 0.25718034333906625|...],
   TargetList = [jai-blava, golkolz, darcaluna, zazgo, brakea, sheeanth, lield,
   aidel, dignarv|...]
?- find_weighted_targets(zhuirzaz, Distances, TargetList)
   Distances = [0.26687880957468313, 0.30606038946586994, 0.3115456467357552,
   0.3784484363820255, 0.3926651652489688, 0.40451883516098475,
   0.40884393844106337, 0.4164695787209433, 0.41727860117671983|...],
   TargetList = [angwispm, engsangu, stermilky, nyax, wistur, ranaqri, thali,
   dorfae, faezab|...]
```

3.8 find_my_best_target(+Name, -Distances, -ActivityList, -CityList, -TargetList), 20 points

This predicate will use all the other restrictions to find possible matching targets together with possible activities in possible cities. So in the end, a glanian will enter her name and will get a list of distances to her matching targets, and activities that can be done in possible cities. We can read each element in four of these lists as follows:

"`Name` and `TargetList[i]` can do `ActivityList[i]` in `City[i]`. This matching is close to the `Name`'s preferences by `Distances[i]`."

The restrictions are as follows (index `i` can be read as for all elements in the list):

1. `Name` and `Target[i]` should not have any previous `oldrelation`.
2. `Name` should either like `City[i]` or be a habitant of `City[i]` (i.e. predicate 3.3), or there should be an `Activity[i]` in `City[i]` that is also in `LikedActivities` of `Name`.
3. `Activity[i]` should not be in `DislikedActivities` of `Name`.
4. `City[i]` should not be in `DislikedCities` of `Name`.
5. `City[i]` should be in `CityList` where `merge_possible_cities(Name, Target, CityList)` is true (i.e. the city should be in the merged possible cities).

6. `Target[i]`'s gender should be in the expected gender list of `Name`.
7. `Target[i]`'s features should be in the tolerance limits (see `dislikes`) of `Name`.
8. The intersection between `Name`'s `DislikedActivities` and `Target[i]`'s `LikedActivities` should not be more than two. In other words, there should not be three or more conflicting activities.

You should return all (Distance, Activity, City, Target) pairs that satisfy the above criterions in respective lists. Use `weighted_glanian_distance(Name, Target, Distance)` for the distance metric. Note that there might be more than one activity in a city for a found match.

Example:

```
?- find_my_best_target(josizar, Distances, ActivityList, CityList, TargetList).
   Distances = [0.5972048350440575, 0.5972048350440575, 0.5972048350440575,
   0.5972048350440575, 0.5972048350440575, 0.5972048350440575,
   0.5972048350440575, 0.5972048350440575, 0.5972048350440575|...]
   ActivityList = [bird_watching, bird_watching, board_gaming, boxing, camping,
   card_game, circus, circus, collecting_leaves|...]
   CityList = [corse_town, viyan, corse_town, viyan, viyan, viyan, corse_town,
   seviliri, seviliri|...]
   TargetList = [tizstarb, tizstarb, tizstarb, tizstarb, tizstarb, tizstarb,
   tizstarb, tizstarb, tizstarb|...]
?- find_my_best_target(anthgall, Distances, ActivityList, CityList, TargetList).
   Distances = [0.2726177800511184, 0.2726177800511184, 0.2726177800511184,
   0.2726177800511184, 0.2726177800511184, 0.2726177800511184,
   0.2726177800511184, 0.2726177800511184, 0.2726177800511184|...]
   ActivityList = [art_gallery, basketball, basketball, bird_watching,
   board_gaming, camping, card_game, circus, circus|...]
   CityList = [honk_gonh, honk_gonh, neu_fork, neu_fork, seaghoul, neu_fork,
   neu_fork, honk_gonh, lonudonu|...],
   TargetList = [gembzynth, gembzynth, gembzynth, gembzynth, gembzynth,
   gembzynth, gembzynth, gembzynth, gembzynth|...].
```

3.9 `find_my_best_match(+Name, -Distances, -ActivityList, -CityList, -TargetList)`, 25 points

This predicate is similar to the previous predicate with some additional constraints. In this predicate, we also take the matching target's preferences into account. The restrictions are as follows (additional constraints are highlighted):

1. `Name` and `Target[i]` should not have any previous `oldrelation`.
2. `Name` should either like `City[i]` or be a habitant of `City[i]` (i.e. predicate 3.3), or there should be an `Activity[i]` in `City[i]` that is also in `LikedActivities` of `Name`.
3. `Target[i]` should either like `City[i]` or be a habitant of `City[i]` (i.e. predicate 3.3), or there should be an `Activity[i]` in `City[i]` that is also in `LikedActivities` of `Target[i]`.

4. `Activity[i]` should not be in `DislikedActivities` of `Name` and `Target[i]`.
5. `City[i]` should not be in `DislikedCities` of `Name` and `Target[i]`.
6. `City[i]` should be in `CityList` where `merge_possible_cities(Name, Target, CityList)` is true (i.e. the city should be in the merged possible cities).
7. `Target[i]`'s gender should be in the expected gender list of `Name`.
8. `Name`'s gender should be in the expected gender list of `Target[i]`.
9. `Target[i]`'s features should be in the tolerance limits (see `dislikes`) of `Name`.
10. `Name`'s features should be in the tolerance limits of `Target[i]`.
11. The intersection between `Name`'s `DislikedActivities` and `Target[i]`'s `LikedActivities` should not be more than two. In other words, there should not be three or more conflicting activities.
12. The intersection between `Name`'s `LikedActivities` and `Target[i]`'s `DislikedActivities` should not be more than two.

You should return all (Distance, Activity, City, Target) pairs that satisfy the above criterions in respective lists. Use the following equation for the distance:

$$\frac{\texttt{wgd(Name, Target, Distance)} + \texttt{wgd(Target, Name, Distance)}}{2} \tag{3}$$

where `wgd` is `weighted_glanian_distance`. Note that there might be more than one activity in a city for a found match.

Examples:

```
?- find_my_best_match(anthgall, Distances, ActivityList, CityList, TargetList).
   Distances = [0.5363785971188019, 0.5363785971188019, 0.5363785971188019,
   0.6186453156203476, 0.6186453156203476, 0.6186453156203476,
   0.6186453156203476, 0.6186453156203476, 0.6186453156203476|...],
   ActivityList = [art_gallery, jet_skiing, jet_skiing, circus, crafting,
   frisbee, jet_skiing, napping, paint|...],
   CityList = [honk_gonh, honk_gonh, lonudonu, lonudonu, lonudonu,
   lonudonu, lonudonu, lonudonu, lonudonu|...],
   TargetList = [kezdark_, kezdark_, kezdark_, azraur, azraur, azraur,
   azraur, azraur, azraur|...].
?- find_my_best_match(nysow, Distances, ActivityList, CityList, TargetList).
   Distances = [0.657633337325202, 0.6699707062805402, 0.6699707062805402,
   0.6699707062805402, 0.6699707062805402, 0.6699707062805402,
   0.6699707062805402, 0.6704309489109601, 0.6704309489109601|...]
   ActivityList = [card_game, card_game, crafting, drink, judo, park, photo,
   bird_watching, camping|...]
   CityList = [venis, venis, venis, seviliri, venis, ansterdum, venis,
   ansterdum, ansterdum|...]
   TargetList = [amamort, narvvine, narvvine, narvvine, narvvine, narvvine,
   narvvine, shadvae, shadvae|...]
```

3.10 Bonus (5 points)

Implement a new predicate which you can decide its arguments that will find the 10 best matches in the whole database. List these 10 matches in a text file `top10.txt` as follows:

```
alper - cagla
alper - mehmet
...
```

Permutations are not allowed in this list (i.e. there should not be an entry `cagla - alper`).

4 Documentation

Please explain what each predicate is for with comments in the code. Codes with no comments might lose points if a predicate is hard to understand.

5 Submission

You will submit two files: `solution.pl`, `feedback.txt`. Your code should be in one file named `solution.pl`. First four lines of your `solution.pl` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

```
% name surname
% student id
% compiling: yes
% complete: yes
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you're doing a partial submission. This whole part must be lowercase and include only the English alphabet. Example:

```
% alper ahmetoglu
% 2012400147
% compiling: yes
% complete: yes
```

We are interested in your feedback about the project. In the `feedback.txt` file, please write your feedback. This is optional, you may leave it empty and omit its submission.

6 Tips and Tricks

Although the project seems long at first glance, it can be done in a reasonable amount of time. Therefore, do not panic.

- Do not rush. First think about the requirements, what you need, how you can solve it in a modular way. If you can verbally describe your needs, you can probably implement it easily.
- Try to formalize the problem, then try to convert the logic formulate to Prolog.

- You can use findall/3, bagof/3 or setof/3.
- You can use extra predicates and it is highly recommended. The ones given above are compulsory.
- If a predicate becomes too complex, either divide it into some predicates or take another approach. Use debugging (through trace/1), approach your program systematically.