Overview

The idea is in each iteration choose a few single mappings[1] (single mapping is like b1:b2::t1:t2) which are satisfy the current solution[2] and maximize the score.
Each iteration creates a valid solution (we accept solution that does not satisfy all entities), so each iterate we add the current solution to the solutions list.
In the end, we will sort this list by some reasonable way (for example, first sort by length of the solution (length=number of entities that matched) and then sort by their score[3].

1)

Choosing the best single mappings

In the beginning of the algorithm, we create a list of all the possible pairs (single mapping) which is updated in each iteration (more in section 2).
So, we iterate over all these single-mappings, and doing this:
For both direction of the single mapping (i.e. if we have b1:b2::t1:t2 we will look also on b2:b1::t2:t1 and summarize their scores), we will:

- Get the entities relations[4] between b1:b2 and t1:t2, if one of them doesn't have relations (b1:b2 or t1:t2), we continue to the other direction.
- Now, we have relations between b1:b2, and relations between t1:t2, so we calculate the score[5] between each relation in b1:b2 to each relation in t1:t2. Call it edges.
- Now we are clustering[6] the relations of b1:b2 and clustering the relations of t1:t2. Then, we take the edges from the previous step, and filter it such that we have only one edge between two clusters. We take the edge with the maximum score.
- Now we are using maximum weighted match of bipartite graph and getting the final edges.
- We take the top 3 edges, which are bigger than some threshold (0.2), and summarize them.

After iterating over all the single mappings, we take the mappings with the highest score (how many we take is depend of 'depth' parameter ,but the default is 4).

2)

Satisfy the current solution

If we already mapped some entities, for example b1->t1 and b2->t2, we need to make sure that the next single mapping that we choose will not break it. For example: b1:b2::t1:t3 is illegal, because b2 is mapped to t2 (in the previous iterate) and now to t3. So only valid solutions will be available.

3)

Solution score

In each iteration we summarize the new entities with the entities that already mapped.
For example, let say in the first iteration we choose: b1:b2::t1:t2, so we add this score (that detailed in section 1). Now, in the second iteration we choose: b1:b3::t1:t3, we will add this score to the previous score, but we will also add the score of: b2:b3::t2:t3.

4)

Get entities relations

Given two entities, we extract the relations between them with the following tools:

Google Auto-Suggest: `^{question}( a)?( an)?( the)? {entity1} (.*) {entity2}$`

With the following questions: "why do", "why is", "why does", "why does it", "why did", "how do", "how is", "how does", "how does it", "how did".

In addition with check for singular/plural.

Quasimodo:

Filtering the DB with subject=entity1 and object=entity2. Taking the top 10 results by the plausibility.

ConceptNet:

Filtering from ConceptNet API with the following fields: "hasProperty", "capableOf", "isA", "usedFor". Taking the top 10 by the weight of conceptNet.

* This doesn't give too many relations...

openIE:

Extract the top 20 (the first page) relations.

`https://openie.allenai.org/search/?arg1={entity1}&arg2={entity2}`

5)

Score between two entities

If one of the relations is in the top 500 n-grams (1-4 words) in Wikipedia, or in manual stopwords, the score is 0.

We calculate the score using ConsineSimilarity on the embedding which taken from sBERT:
https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html

The result is score in range (0,1]. The highest is more similar.

If the score is lower then 0.2, we treat it as 0.

6)

Clustering

The clustering is done by using the cosine similarity (of the embedding which calculate with sBERT) and distance threshold. The default distance threshold is 0.8 (1 will put all in one cluster, 0 will put all in separate). This distance was chosen by trial and error.

The base of the algorithm was taken from here: https://github.com/UKPLab/sentence-transformers/blob/master/examples/applications/clustering/agglomerative.py