Mapping

```
mapping( base: List[string], target: List[string] ) -> List[str]:
  --------------------------------
  // assuming len(base) == n, len(target) == m
  // there are ((n choose 2) * (m choose 2) * 2) pairs
  possible_pairs = get_all_possible_pairs(base, target)

  // here we going to store the entities that already mapped.
  // the value in index i in both lists will be the map between them.
  // it is clear that both must be in the same length.
  base_already_map, target_already_map = [], []

  while len(base_already_map) < min(len(base), len(target)):
    // updating the possible pairs according to the entities that already mapped
    // the idea is to not break the entities that already mapped.
    update_possible_pairs(possible_pairs, base_already_map, target_already_map)

    // we want the pair with the best score.
    // the meaning of pair is for example: earth→electrons AND sun→nucleus.
    res = get_best_pair_mapping(possible_pairs)

    if res["score"] > 0:
      // updating the already mapped lists.
      // res["base"][0] → res["target"][0], res["base"][1] → res["target"][1]
      update_list(base_already_map, res["base"])
      update_list(target_already_map, res["target"])
    else:
      // no map found at all.
      break

  // suggestions for entities which are not mapped.
  base_suggestions = get_suggestions(base_not_mapped, base_already_map, target_already_map)
  target_suggestions = get_suggestions(target_not_mapped, target_already_map, base_already_map)
  --------------------------------
  return [f"{b} → {t}" for b, t in zip(base_already_map, target_already_map)]
```

```
get_best_pair_mapping( pairs: List[List[Tuple[string]]] ) -> List[Tuple[string]]:
  mapping = []
  --------------------------------
  for pair in pairs:
    // pair is something like: [(earth, sun), (electrons, nucleus)]
    base_edge, target_edge = pair
    mapping.append(pair, get_score(base_edge, target_edge))
  --------------------------------
  return sorted(mapping, key=lambda x: [1], reverse=True)[0]
```

Clustering + score

```
1.  get_score( e_1: tuple, e_2: tuple):
2.    --------------------------------
3.    score = 0
4.    // we count both directions, for example for mapping earth→electrons, sun→nucleus
5.    // we will count (earth:sun,electrons:nucleus) and (sun:earth,nucleus:electrons)
6.    for i in range(2):  // direction
7.      // e_1 and e_2 will flip in the second iteration (direction..)
8.      props_1 = get_edge_props( e_1 )  // List[str]
9.      props_2 = get_edge_props( e_2 )  // List[str]
10.
11.     // this will create a full bipartite graph between props_1 and props_2
12.     similarity_edges = get_edges_weights( props_1, props_2 ) // List[Tuple[str, str, float]]
13.
14.     // clustering is using AgglomerativeClustering of sklearn.cluster
15.     // https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
16.     // distance_threshold → how close the props in the cluster
17.     clusters_props_1 = clustering( props_1, distance_threshold ) // Dict[int, List[str]]
18.     clusters_props_2 = clustering( props_2, distance_threshold ) // Dict[int, List[str]]
19.
20.     // between every two clusters (from the opposite side of the bipartite) we will take
21.     // only one edge, which will be the one with the maximum weight.
22.     clusters_edges = get_clusters_edges( similarity_edges, clusters_props_1, clusters_props_1 )
23.
24.     // we want the maximum-weight of full bipartite matching
25.     // we will use networkx algorithm of minimum_weight_full_matching
26.     // https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.minimum_weight_full_matching.html
27.     best_matching = maximum_weight_full_matching( clusters_edges )
28.     score += sum([edge[2] for edge in best_matching])
29.   --------------------------------
30.   return score
```

```
1.  get_edge_props( subject: string, object: string ) -> List[str]:
2.    --------------------------------
3.    props_1 = get_props_from_quasimodo( subject, object, n_largest = 10 )  // sorted by plausibility
4.    props_2 = get_props_from_google( subject, object )  // why|how do|does|did
5.    props_3 = get_props_from_conceptnet( subject, object )  // sorted by concept-net weights
6.    --------------------------------
7.    return props_1 + props_2 + props_3
```

```
1.  get_edges_weights( props_edge_1: List[string], props_edge_2: List[string] ):
2.    --------------------------------
3.    edges = []
4.    for p_1 in props_edge_1:
5.      for p_2 in props_edge_2:
6.        // similarity is calculated by cosine-similarity.
7.        // https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html
8.        edges.append( (p_1, p_2, similarity( p_1,p_2 )) )
9.    --------------------------------
10.   return edges
```

Suggestions

```
get_suggestions( base_not_mapped: List[string], base_already_map: List[string],
                                                target_already_map: List[string]  ) -> List[str]:
  ---------------------------------
  suggests = {}
  // we will iterate on each entity that not mapped
  for base_not_mapped_entity in base_not_mapped:
    suggestions_list = []
    // we need all the relations between the entity to the other entities in the same domain
    for idx, base_entity in enumerate(base_already_map):
      // we need the relations in both directions
      relations1 = get_entities_relations(base_entity, base_not_mapped_entity)
      relations2 = get_entities_relations(base_not_mapped_entity, base_entity)

      // now we will iterate over the relations.
      // we know that base_entity -> target_already_map[idx] so we will use it.
      for relation in relatinos1 + relations2:
        // for now, the suggestions are based on google autosuggest
        suggestions_list.extend(get_entity_suggestions(target_already_map[idx], relation))

    // it can be dozens of suggestions, so we will filter some of them
    suggests[base_not_mapped_entity] = get_best_match(base_not_mapped_entity, suggestions_list)
  ---------------------------------
  return suggests
```

```
get_best_match( entity: string, suggestions: List[string], n_best: int) -> List[string]:
  ---------------------------------
  best = []
  for suggest in suggestions:
    best.append(suggest, get_similarity_score(suggest, entity))
  ---------------------------------
  return sorted(best, key=lambda x: x[1], reverse=True)[:n_best]
```

```
get_similarity_score( entity1: string, entity2: string) -> float:
  ---------------------------------
  // we extract the props of the single entity from Quasimodo, google and conceptNet.
  props1 = get_entity_props(entity1)
  props2 = get_entity_props(entity2)

  // we want the maximum weights match between the props of entity1 and the props of entity2
  // assuming this return the edges sorted by their score, we want the average of the best 3
  similarity_edges = get_maximum_weighted_match(props1, props2)
  ---------------------------------
  return similarity_edges[:3]
```