**Mapping**

1. mapping( **base:** List[string], **target:** List[string] ) -> List[str]:
2. --------------------------------
3. // assuming len(base) == n, len(target) == m
4. // there are ((n choose 2) \* (m choose 2) \* 2) pairs
5. possible\_pairs = get\_all\_possible\_pairs(base, target)
7. // here we going to store the entities that already mapped.
8. // the value in index ***i*** in both lists will be the map between them.
9. // it is clear that both must be in the same length.
10. base\_already\_map, target\_already\_map = [], []
12. while len(base\_already\_map) < min(len(base), len(target)):
13. // updating the possible pairs according to the entities that already mapped
14. // the idea is to not break the entities that already mapped.
15. update\_possible\_pairs(possible\_pairs, base\_already\_map, target\_already\_map)
17. // we want the pair with the best score.
18. // the meaning of pair is for example: earth🡪electrons **AND** sun🡪nucleus.
19. res = get\_best\_pair\_mapping(possible\_pairs)
21. if res["score"] > 0:
22. // updating the already mapped lists.
23. // res["base"][0] 🡪 res["target"][0], res["base"][1] 🡪 res["target"][1]
24. update\_list(base\_already\_map, res["base"])
25. update\_list(target\_already\_map, res["target"])
26. else:
27. // no map found at all.
28. break
29. --------------------------------
30. return [f"{b} 🡪 {t}" for b, t in zip(base\_already\_map, target\_already\_map)]
31. get\_best\_pair\_mapping( **pairs:** List[List[Tuple[string]]] ) -> List[Tuple[string]]:
32. mapping = []
33. --------------------------------
34. for pair in pairs:
35. // pair is something like: [(earth, sun), (electrons, nucleus)]
36. base\_edge, target\_edge = pair
37. mapping.append(pair, get\_score(base\_edge, target\_edge))
38. --------------------------------
39. return sorted(mapping, key=lambda x: [1], reverse=True)[0]

**Clustering + score**

1. get\_score( : tuple, : 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. // and will flip in the second iteration (direction..)
8. = // List[str]
9. = // List[str]
10. // this will create a full bipartite graph between and
11. = // List[Tuple[str, str, float]]
13. // clustering is using AgglomerativeClustering of sklearn.cluster
14. // <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html>
15. // 🡪 how close the props in the cluster
16. = // Dict[int, List[str]]
17. = // Dict[int, List[str]]
18. // between every two clusters (from the opposite side of the bipartite) we will take
19. // only one edge, which will be the one with the maximum weight.
20. clusters\_edges = get\_clusters\_edges( , , )
21. // we want the maximum-weight of full bipartite matching
22. // we will use networkx algorithm of minimum\_weight\_full\_matching
23. // https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.minimum\_weight\_full\_matching.html
24. best\_matching = maximum\_weight\_full\_matching( clusters\_edges )
25. score += sum([edge[2] for edge in best\_matching])
26. --------------------------------
27. return score
28. get\_edge\_props( : string, : string ) -> List[str]:
29. --------------------------------
30. = // sorted by plausibility
31. = // why|how do|does|did
32. = // sorted by concept-net weights
33. --------------------------------
34. return
35. get\_edges\_weights( : List[string], : List[string] ):
36. --------------------------------
37. edges = []
38. for in :
39. for in :
40. // similarity is calculated by cosine-similarity.
41. // https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html
43. --------------------------------
44. return