

Final Project

Project Code

<https://github.com/svilag/SI507-Final>

Data Sources

The HTML data is stored in on <https://wgi.org/>. The data included is divided by competition, and includes scores for each group that participated in that competition.

URLs:

```
[  
  "https://wgi.org/percussion/perc-scores-2022/",  
  "https://wgi.org/percussion/perc-scores-2020/",  
  "https://wgi.org/percussion/2019-perc-scores/",  
  "https://wgi.org/percussion/2018-perc-scores/",  
  "https://wgi.org/winds/winds-scores-2022/",  
  "https://wgi.org/winds/winds-scores-2020/",  
  "https://wgi.org/winds/2019-winds-scores/",  
  "https://wgi.org/2018-winds-scores/",  
  "https://wgi.org/color-guard/2022-scores/",  
  "https://wgi.org/color-guard/scores-2020/",  
  "https://wgi.org/color-guard/scores-2019/",  
  "https://wgi.org/color-guard/scores-2018/"  
]
```

Each link in the above list contains about 30-40 links to pages with score data, and about 30-40 links to detailed recap pages. Each url in the list was parsed and the urls for scores and recaps pages grabbed, stored as part of the Competitions class, then those pages were parsed to grab the scores.

The scores pages include a table with performance scores for each group, divided by group class level (example score page: https://wgi.org/wp-content/uploads/wgi_events/static_scores/2022/scores_San_Bernardino_Perc_Finals.html).

The recap pages contain similar data, but in a more complex table with breakdowns by category with the names of judges for each category listed. Theses scores are also divided by group class level (example recaps page: <https://recaps.competitionsuite.com/24c4c8ec-a300-4737-816b-e78dc0a3221c.htm>).

Records

The JSON for each Competition is written to [./data/competitions.json](#). The JSON for each Group is written to [./data/groups.json](#).

Competitions: There are 390 competitions.

Each Competition object stores:

- name of the competition
- competition date
- url to the scores page so the scores can be parsed
- url to the recaps so the recaps can be parsed
- list of group names that participated in the competition
- dictionary of groups and their score for that competition

```
{
  "title": "competition_name",
  "date": "competition_date",
  "scores": "url_to_scores_page",
  "recap": "url_to_recap_page",
  "groups": ["group_name", "..."],
  "scores_by_group": {"group_name": "score"}
}
```

Groups: There are 946 groups.

Each Group object stores:

- type of group (percussion, winds, color guard): groups of the same type compete against each other
- name of the group
- class level: groups of the same class level are scored with each other
- location the group is from
- dictionary of competitions the group participated in and the score received

```
{
  "group_type": "group_type",
  "name": "group_name",
  "class_level": "class_level",
  "location": "location",
  "competitions": {"comp_name": "score"}
}
```

Caching

As each url is parsed it is stored in a cache by calling the function `get_from_cache()`.

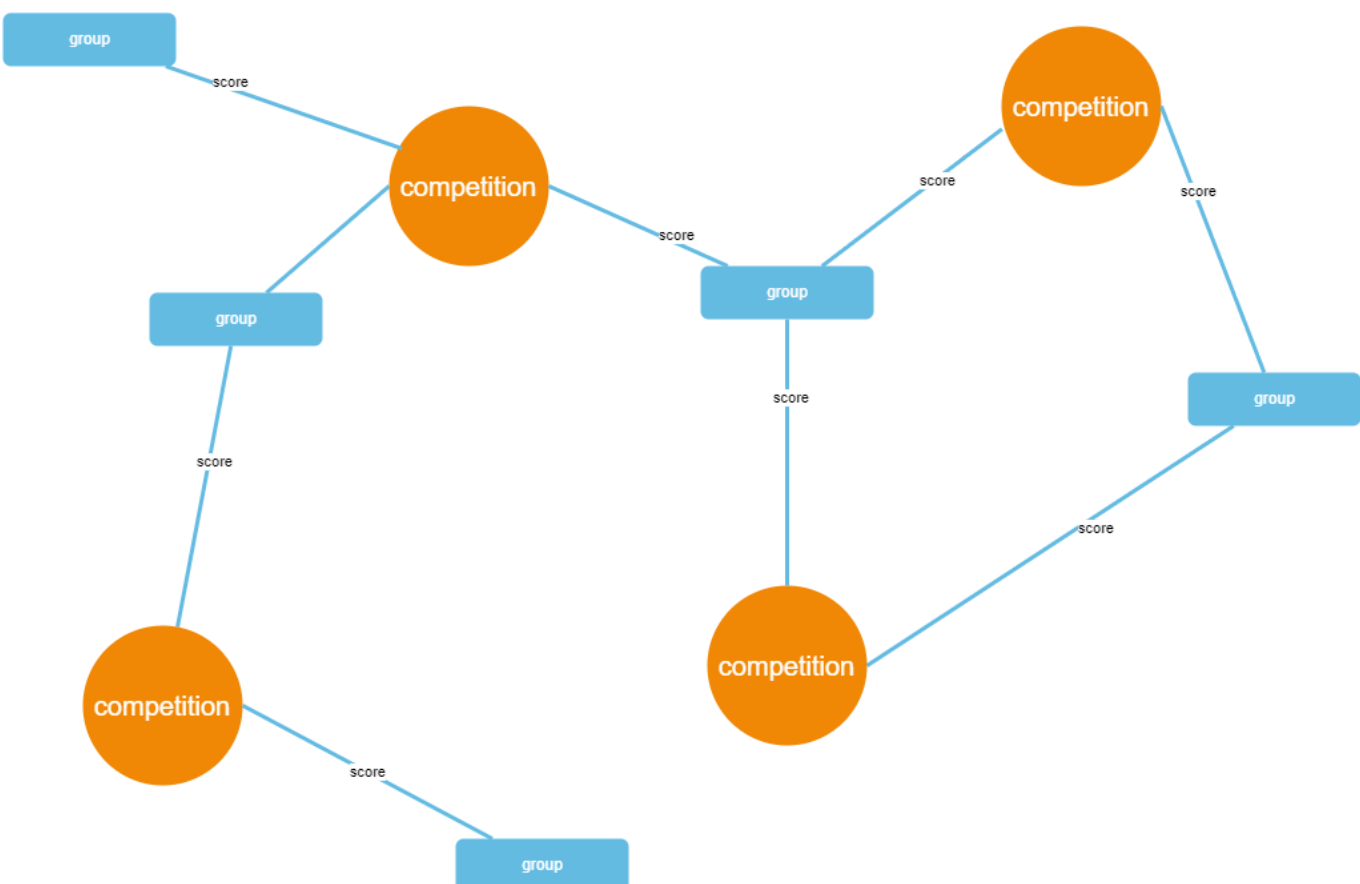
```
def read_cache() -> dict:
    """ reads the cache file
        returns the cache
    """
    cache = read_json(CACHE_PATH)
    # logger.info("Reading from cache...")
    return cache
```

```
def check_cache(url:str) -> str:
    """ checks cache for url.
        if url not in cache, adds it and rewrites the cache.
        returns html content
    """
    cache = read_cache()
    if url not in cache.keys():
        cache[url] = get_content(url)
        write_json(CACHE_PATH, cache)
    #     logger.info("Writing %s to cache...", url)
    # logger.info("Fetching %s from cache...", url)
    return cache[url]
```

Data Structure

The data will be stored in a graph. The initial proposal indicated a tree would be used, but upon interacting with the data further, a graph seems more appropriate.

Each vertex in the graph will be either a competition or a group, with the edge between each group and competition being the score the group received for the competition.



```
class Vertex():
    def __init__(self, key):
        self.id = key
        self.connected_to = {}

    def add_neighbor(self, nbr, weight=0):
```

```

        self.connected_to[nbr] = weight

    def get_id(self):
        return self.id

    def get_weight(self, nbr):
        return self.connected_to[nbr]

    def get_connections(self):
        return self.connected_to.keys()

    def __str__(self):
        return f"{str(self.id)} is connected to {str([x.id for x in self.connected_to])}"

class Graph():
    def __init__(self):
        self.vert_list = {}
        self.num_verts = 0

    def add_vertex(self, key):
        self.num_verts += 1
        new_vert = Vertex(key)
        self.vert_list[key] = new_vert
        return new_vert

    def get_vertex(self, n):
        if n in self.vert_list:
            return self.vert_list[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vert_list

    def add_edge (self, f, t, weight=0):
        if f not in self.vert_list:
            nv = self.add_vertex(f)
        if t not in self.vert_list:
            nv = self.add_vertex(t)
        self.vert_list[f].add_neighbor(self.vert_list[t], weight)

    def get_vertices(self):
        return self.vert_list.keys()

    def __iter__(self):
        return iter(self.vert_list.keys())

groups = done_cleaning # Group objects after cleaning
competitions = all_competitions # Competition objects

wgi_graph = Graph()

for comp in competitions:

```

```
for group in comp.groups:
    wgi_graph.add_edge(f"{comp.title}, {comp.date}", group.name,
comp.scores_by_group[group.name])
    wgi_graph.add_edge(group.name, f"{comp.title}, {comp.date}",
comp.scores_by_group[group.name])
```