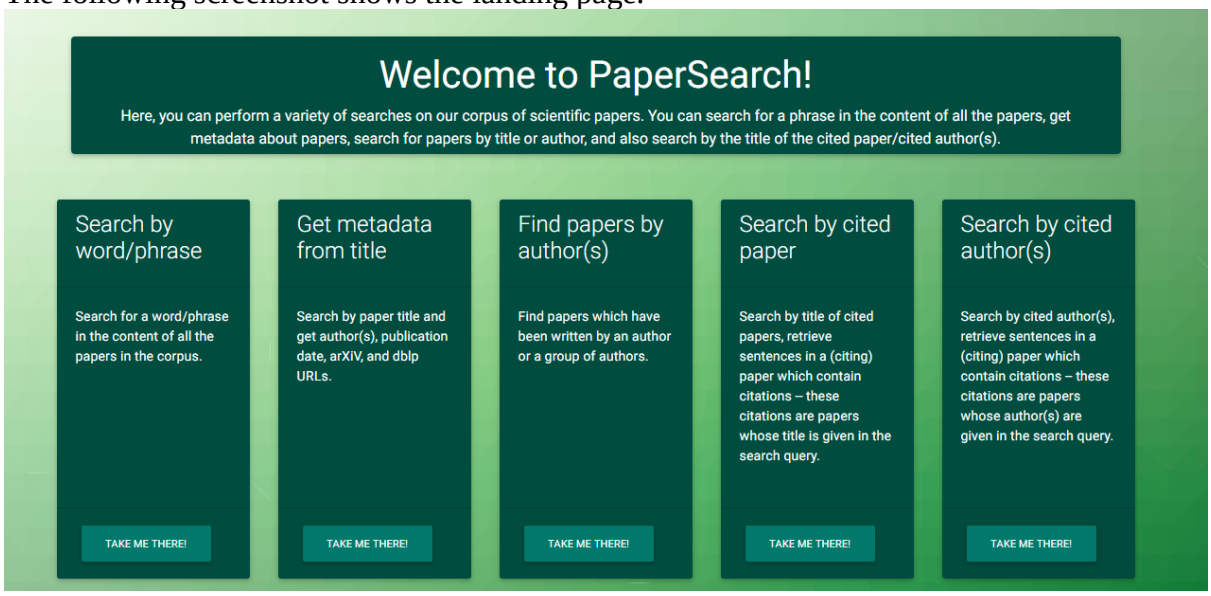
Django and Python Process and Code for PaperSearch

Languages/tools used: Apache Solr, Python 3.5, Django, HTML, CSS3, MDBootstrap,

PREREQUISITES: Solr indices should exist as described in the solr_steps.pdf file. papers, arxiv_metadata, metadata, references are the names of the 4 Solr indices, which are assumed to exist.

So now that the Solr configuration is done, and the data has been inserted into Solr, it's time for querying from the webpage (created using Django) and getting the desired results from Solr into Python, and from there into the HTML.
The website for PaperSearch can be reached at **http://132.230.150.9**, which automatically redirects to **http://132.230.150.9/searchengine/.**
The following screenshot shows the landing page.



As you can see, there are 5 options in the above main page, the html is in the template **index.html.**
These 5 options are given again below. Clicking on the button below each of them takes you to the corresponding search page.

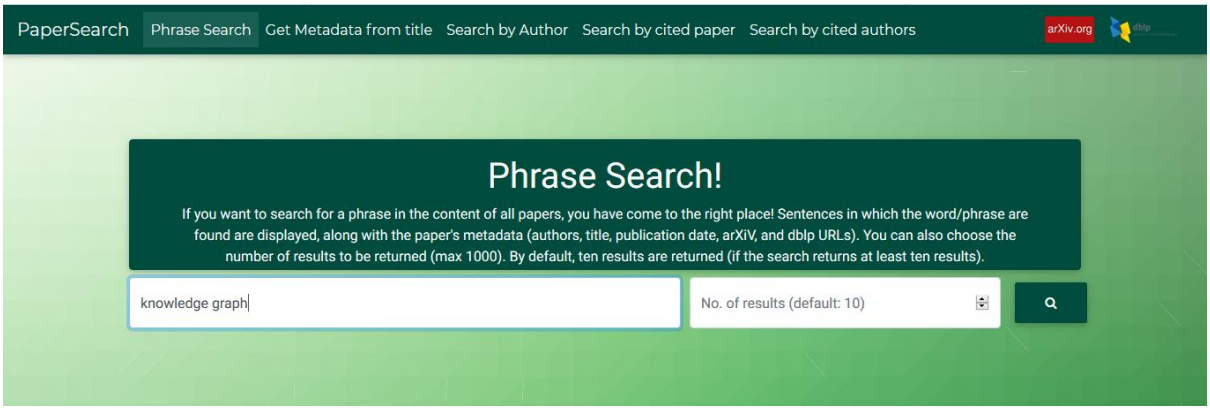
1. Search for a word/phrase in the content of all the papers in the corpus.
2. Search by paper title and get author(s), publication date, arXiV, and dblp URLs.
3. Find papers which have been written by an author or a group of authors.
4. Search by title of cited papers, retrieve sentences in a (citing) paper which contain citations -- these citations are papers whose title is given in the search query.
5. Search by cited author(s), retrieve sentences in a (citing) paper which contain citations -- these citations are papers whose author(s) are given in the search query.

The Django and the front-end explanation are given below for the first option 'phrase search' and it is similar for the others. So, only the screenshots are given for these options, sans (much) explanation. The back-end logic is explained later on, after the front-end explanation.

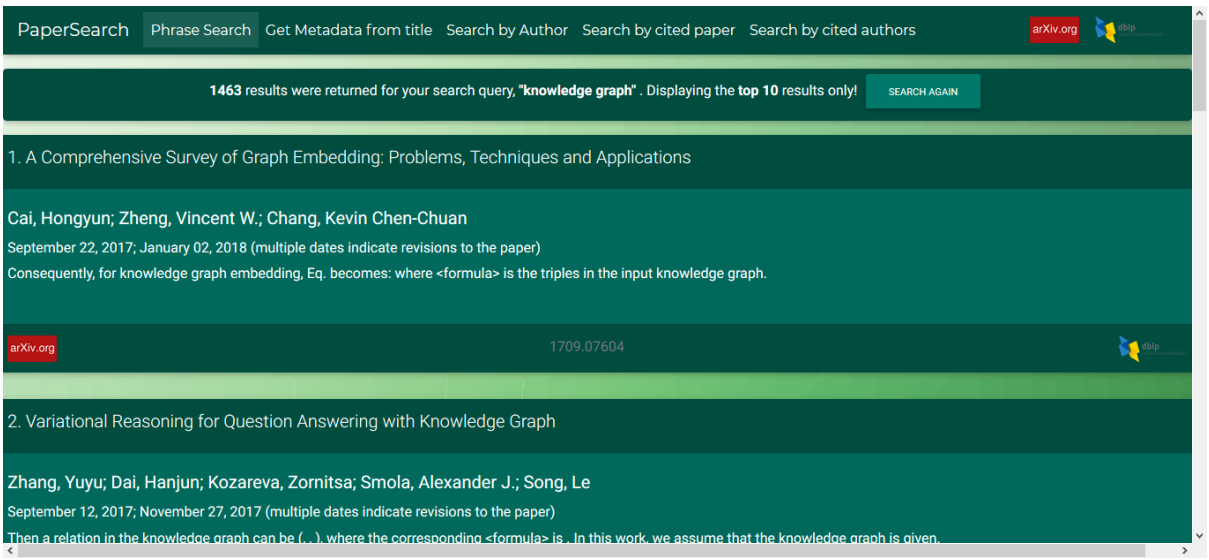
DJANGO AND FRONT-END EXPLANATION

1. **Phrase Search**

   The search page is given below, the template is phrasesearch.html. The page has a search box for the phrase as well as another option for 'no. of results' (10 by default if no value is given). The function phrase_search in views.py is called whenever this page is displayed. It does the field validation and calls the search_sentences Python function in the django_paper_search module. search_sentences returns a list of results, with all the fields including title, authors, arxiv url, dblp url (if it exists), and arxiv identifier.



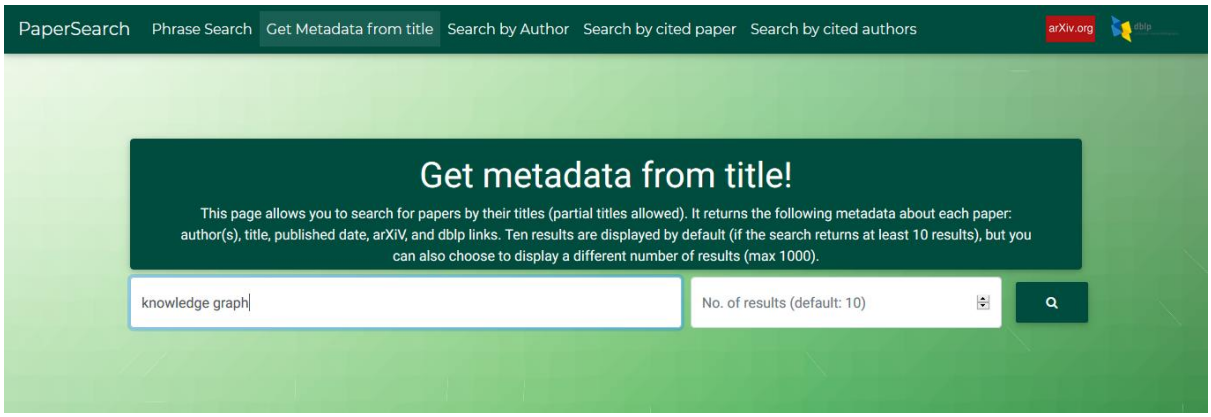
   The views function phrase_search finally renders the results in the template phrasesearchresults.html.
   The search results are shown in the following screenshot.



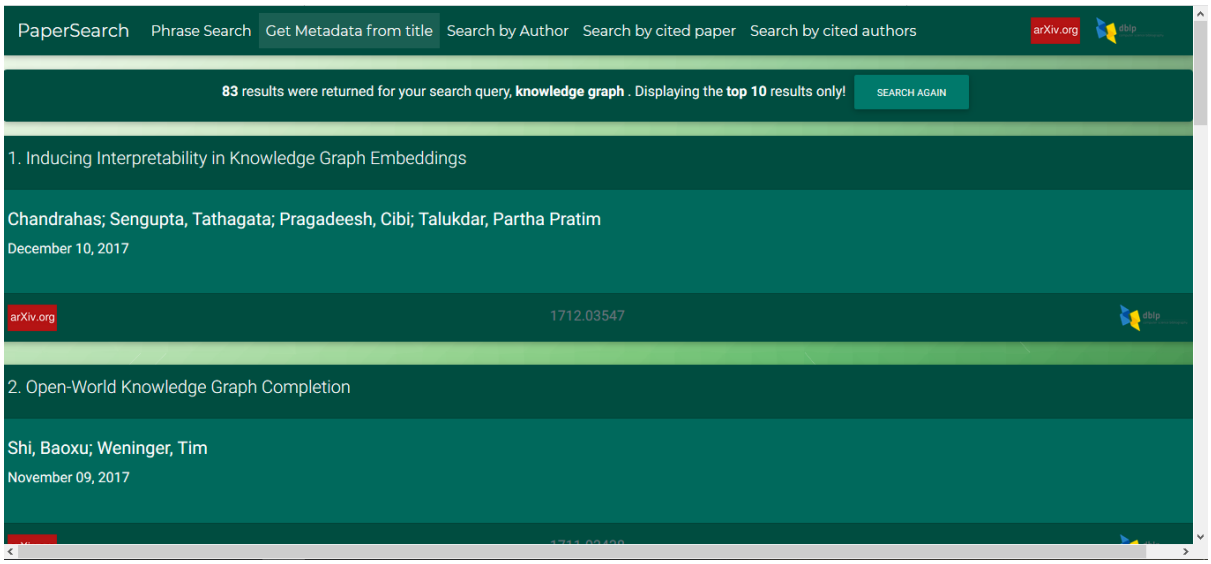
   This template receives the results (a Python list) from the view, and iterates through them using Django template tags and filter, and renders all the fields in separate CSS 'cards' for each result (using MDBootstrap's version of cards). Each card also has a DBLP and Arxiv url link (logo links). While arxiv links always exist, the dblp url may or may not exist for a result. This is checked in template logic, and if there is no dblp url, the logo link is disabled.

2. **Get metadata from title**
   The Django explanation is similar to that for phrase search, the templates are titlesearch.html and titlesearchresults.html. The corresponding view function is metadatatitle_search.

Search:

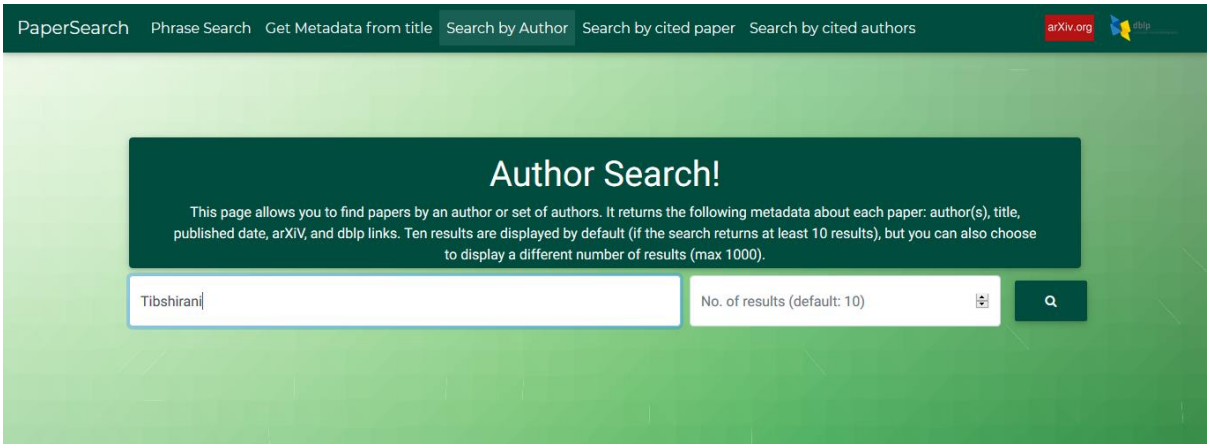


Search Results:

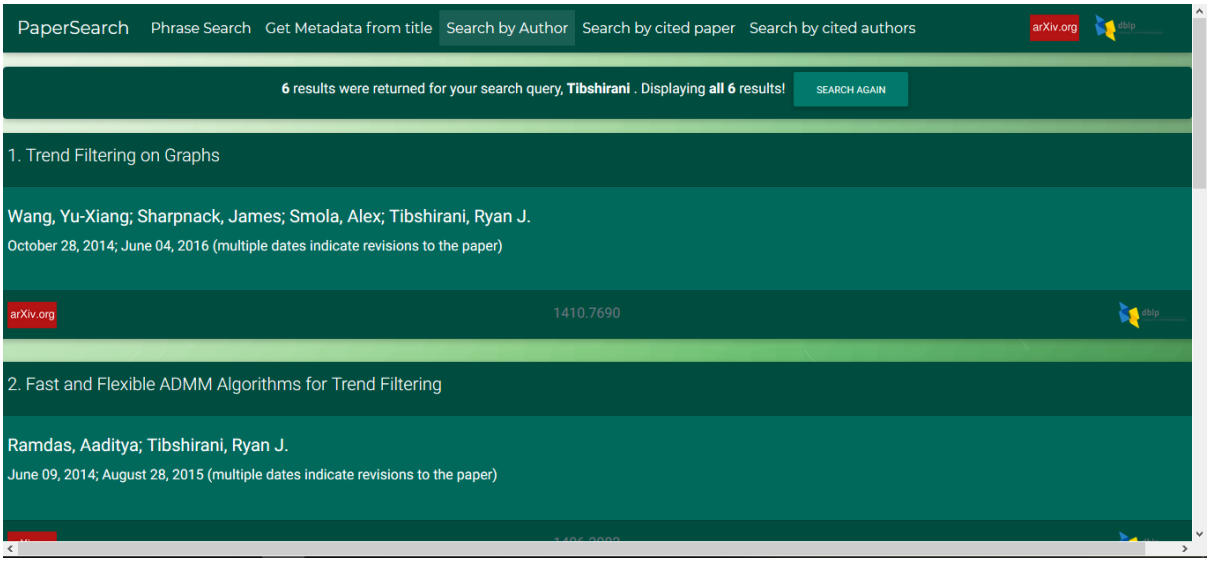


3. **Author Search**

Search: **Multiple authors need to be separated by semicolons.** the templates are authorsearch.html and authorsearchresults.html. The corresponding view function is author_search.



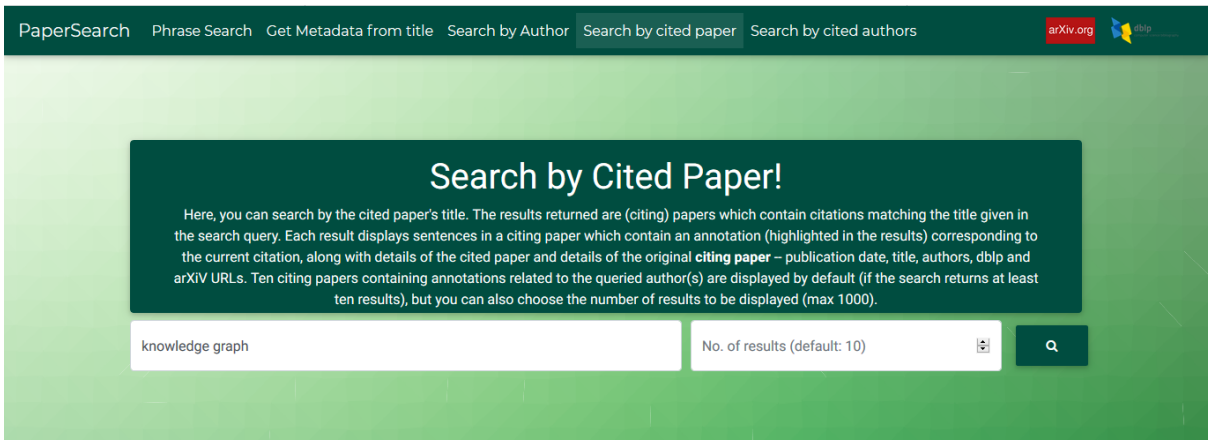
Search Results:



4. **Search by Cited Paper**

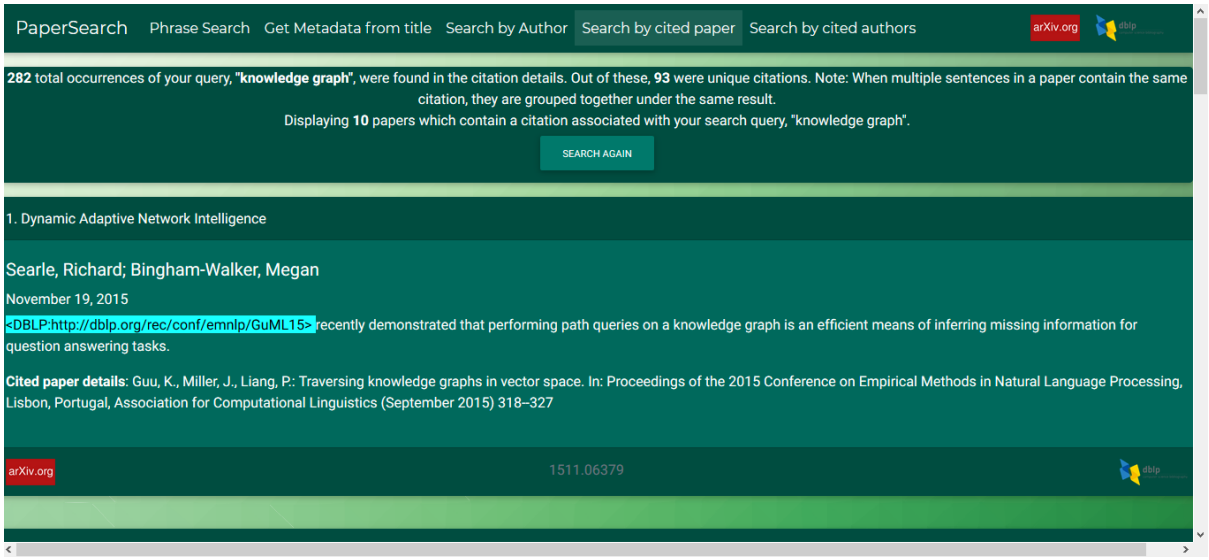
It allows the user to search by the cited paper's title. The results returned are (citing) papers which contain citations matching the title given in the search query. Each result displays sentences in a citing paper which contain an annotation (highlighted in the results) corresponding to the current citation, along with details of the cited paper and details of the original **citing paper** -- publication date, title, authors, dblp and arXiV URLs. The templates are citedpapersearch.html and citedpapersearchresults.html. The corresponding view function is cited_paper_search.

Search:



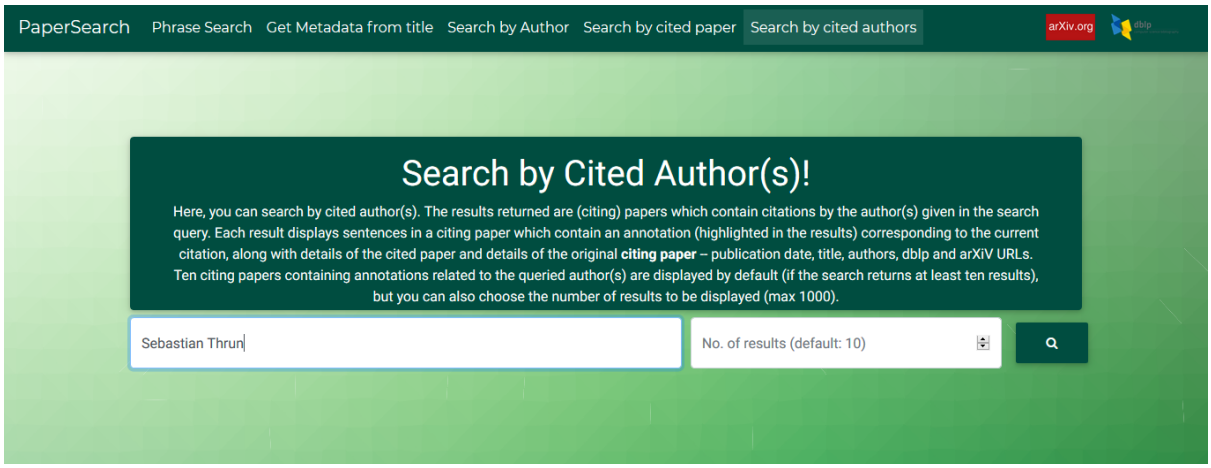
Search Results:

The search results include the details of the cited paper AND the citing paper. When multiple sentences in a paper contain the same citation, they are grouped together under the same result.

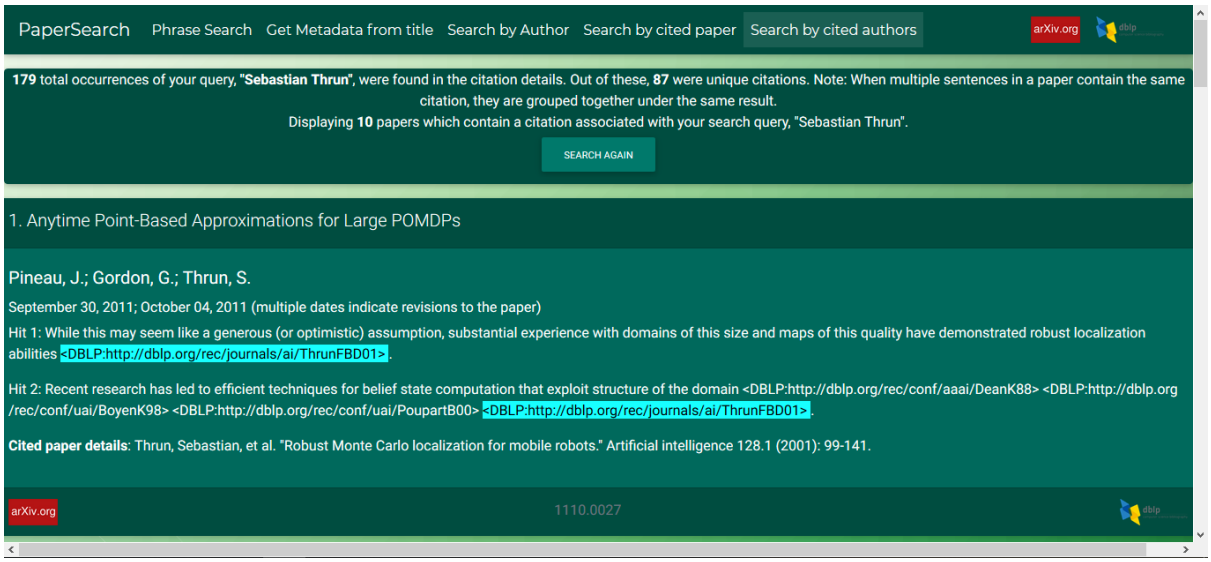


5. **Search by cited author:**
It allows the user to search by the cited paper's authors. The results returned are (citing) papers which contain citations matching the title given in the search query. Each result displays sentences in a citing paper which contain an annotation (highlighted in the results) corresponding to the current citation, along with details of the cited paper and details of the original **citing paper** -- publication date, title, authors, dblp and arXiV URLs
The templates are citedauthorsearch.html and citedauthorsearchresults.html. The corresponding view function is cited_author_search.

Search:



Search Results:
The search results include the details of the cited paper AND the citing paper. When multiple sentences in a paper contain the same citation, they are grouped together under the same result.



NOTE: The published date and the authors are normalized in views.py before they are displayed in the templates.

BACK END FUNCTIONS CALLED BY THE DJANGO VIEW FUNCTIONS:
Module: django_paper_search.py
There are a number of functions which are used (by different options: there's a lot of overlap especially among functions search_solr, parse_json and add_query_type).
Here are the functions:
search_sentences, search_references, search_authors, search_meta_titles, num_rows_input, add_query_type, search_solr, parse_json, parse_sentence_json, parse_metadata_json, parse_refs_json
The following common functions are explained first: add_query_type, search_solr, parse_json.

search_solr and add_query_type:
search_solr is called in different options to query different collections. It takes the fields shown below: search_filed is the field in the solr index which is searched (using the 'df' parameter: see Solr querying tutorial for more about this). The query_type can be one of three things: 'exact', 'proximity' or 'and'. These correspond to 3 different kinds of queries using Solr's standard query parser.
The add_query_type translates the query_type from plain English to the format Solr expects and returns it in this format to search_solr.
1. Exact query: surrounded by quotes. "query"
2. Proximity query: surrounded by quotes, and a tilde symbol and a number n, which indicates that there can be n words between the words within the quotes. This is useful to search for cases where the second word in the phrase occurs before the first phrase. The number n is set to the no. of words in the user's search phrase. The proximity_authors query type defined in the add_query_type function allows a higher proximity than proximity_title. This is to allow users to search for last name of the first author and the first name of the 4th author, for example.
3. And query: this is used in the case where there is a search for more than 1 author. It creates a list of proximity queries and then does an AND operation on them.

Once the query_type is returned to search_solr, it builds a solr url using the requrests library. All the params which we want to pass to Solr -- q, rows, df – are added to a dictionary of params, which can be passed along with the url in request's beautiful get method. This sends the GET request to SOLR and gets SOLR's json response in data. This response is then sent to parse_json.

```
def search_solr(query, num_rows, collection, search_field, query_type):
    """ Creates a URL to call Solr along with the search query, search
field
    and number of rows as parameters, and sends a GET request to SOLR. It
    then calls the parse_json func to parse the json, and returns results
    from that function."""
    solr_url = 'http://localhost:8983/solr/' + collection + '/select'
    query = add_query_type(query, query_type)
    url_params = {'q': query, 'rows': num_rows, 'df': search_field}
    solr_response = requests.get(solr_url, params=url_params)
    if solr_response.ok:
        data = solr_response.json()
        return parse_json(data, collection)
    else:
        print("Invalid response returned from Solr")
        sys.exit(11)
```

add_query_type:

```
def add_query_type(query, query_type):
    """ Returns the query based on the query type (exact or proximity)
    required for different searches. """
    if query_type == 'exact':
        query = '"' + query + '"'
    elif query_type == 'proximity_authors':
        # Allow for the words in the query to be in a different order. There may also be an
        # 'and' between the words in the file/index. This also allows for search by last
name
        # of 1 author and full name of third author.
        query = '"' + query + '"' + '~' + str(len(query.split())+8)
    elif query_type == 'proximity_title':
        # Allow for the words in the query to be in a different order. There may also be an
        # 'and' between the words in the file/index. This also allows for search by last
name
        # of 1 author and full name of third author.
        query = '"' + query + '"' + '~' + str(len(query.split()))

    elif query_type == 'and':
        # query is a list, authors search
        query = ['"' + name + '"~' + str(len(name.split())) for name in
query]
        query = ' AND '.join(query)
    return query
```

parse_json:
This function is used to parse the JSON returned by the SOLR query. It calls different functions: parse_refs_json, parse_metadata_json, parse_arxiv_metadata_json, parse_sentence_json, based on which Solr index the data is obtained from. Parse_sentence_json is given below, the others are pretty similar.

```
def parse_sentence_json(data):
    """ Function to parse the json response from the papers collection
    in Solr. It returns the results as a list with the sentence, file name
    and title."""
    # docs contains sentence, fileName, id generated by Solr
    docs = data['response']['docs']
    # Create a list object for the results with sentence, fileName and title
    results = [[docs[i].get('sentence')[0], docs[i].get('fileName')]
               for i in range(len(data['response']['docs']))]
    return results
```

Option 1: Search for a phrase in the content of all papers.
search_sentences: it is called by the Django view phrase_search.
Takes user's query as input, finds all sentences with the given phrase, then finds the title, authors and url of the paper from the arxiv_metadata and metadata indices. It also gets the results and normalizes it so that correct errors messages are displayed, and fields are displayed in the right format.
To do this, it first calls search_solr on the papers index. This gets the arxiv identifier (filename in Solr) and the sentence for each result. Results is a list because search_solr, in this case, calls parse_sentence json to parse the JSON and put it in a list.

```
    results, query, num_results = search_solr(query, num_rows,
                                              'papers', 'sentence', 'exact')
```

If there are results, for each result, a query against the arxiv_metadata Solr index is made to get the title, authors, published date and the arxiv url.

```
res_arxiv, _, _ = search_solr(arxiv_identifier, 1,
                              'arxiv_metadata', 'arxiv_identifier', 'exact')
```

The parse_arxiv_metadata_json function is shown below (called by search_solr for the arxiv_metadata collection.

```
def parse_arxiv_metadata_json(data):
    """ Function to parse the json response from the metadata or the
    arxiv_metadata collections in Solr. It returns the results as a
    list with the sentence, file name and title."""
    # docs contains authors, title, id generated by Solr, url
    docs = data['response']['docs']
    # NOTE: there are records without authors and urls. This is why the
    # get method is always used to get the value instead of getting the value
    # by applying the [] operator on the key.
    results = [[docs[i].get('title'), docs[i].get('authors'),
                docs[i].get('url'), docs[i].get('arxiv_identifier'),
                docs[i].get('published_date')]
                for i in range(len(data['response']['docs']))]
    return results
```

The 'metadata' index, which has the dblp url, is also queried. This index, of course, is built from the 'meta' files. These files have a lot of missing values.

```
res_dblp, _, _ = search_solr(arxiv_identifier, 1,
                             'metadata', 'arxiv_identifier', 'exact')
```

These are joined together, the different field values are normalized, and the results are returned, along with the number of results num_results.

Option 2:
Search by paper title and get author(s), publication date, arXiV, and dblp URLs.
It returns all metadata (title, authors, url) when a partial or complete title is given in the user query.
It makes a search on first the arxiv_metadata index on 'title', and then gets the dblp url for each of these results (based on a join with the arxiv identifier)

```
    results, query, num_results = search_solr(query, num_rows,
                                              'arxiv_metadata', 'title',
                                              'exact')
```

```
    for result in results:
            arxivid = result[3]
            # This should return only 1 row (exact match on arxiv_identifier
            res_dblp, _, _ = search_solr(arxivid, 1, 'metadata',
'arxiv_identifier', 'exact')
```

These results are then returned.

Option 3:
Find papers which have been written by an author or a group of authors.
It returns all metadata (title, authors, url) when names of 1 or more authors are given in the user query. This is very similar to option 2. The same fields are returned.

```
    results, query, num_results = search_solr(query, num_rows,
                                              'arxiv_metadata', 'authors',
                                              'and')
    for result in results:
            arxivid = result[3]
            # This should return only 1 row (exact match on arxiv_identifier
            res_dblp, _, _ = search_solr(arxivid, 1, 'metadata',
'arxiv_identifier', 'exact')
```

Options 4 and 5:
Search by title of cited papers, retrieve sentences in a (citing) paper which contain citations -- these citations are papers whose title is given in the search query.
Search by cited author(s), retrieve sentences in a (citing) paper which contain citations -- these citations are papers whose author(s) are given in the search query.

Both these options query the references Solr index on the details field and are defined in the same function search_references. Both are Solr proximity searches, but the amount of proximity is different (see add_query_type).

```
    if search_type == 'title':
        # Return all rows to count no. of citations.
        results, query, num_results = search_solr(query, 100000,
                                              'references', 'details',
                                              'proximity_title')

    if search_type == 'authors':
        # Return all rows to count no. of citations.
        results, query, num_results = search_solr(query, 100000,
                                                  'references', 'details',
                                                  'proximity_authors')
```

In the above function calls, the user's query (author/title of cited papers in the refs files) needs to get all a certain number of from the references index. Solr doesn't have the capability of getting 'all' results, the only way is to give a very high number in numrows: much higher than the no. of results possible in the use case. Now these are the total no. of citations found. However, different refs files WILL contain the same citation, so what we are really interested in is the no. of unique citations.

Note that the results are of the form [(annotation, details), (annotation, details),…], where details is a field in the refs file containing authors, title and other publication details.
The refs files are not standardized at all, there are many cases where the same annotation has different details associated with it. But while finding the unique citations, the value of the details is not taken into account, I take into account only the value of the annotation.
To do this, make an Ordered Dict with the annotation as key and details as the value.

```
    unique_citations = OrderedDict((result[0], result[1]) for result in
results)
    num_unique_citations = len(unique_citations)
    # Convert unique_citations back into a list -- a list of tuples so that we can slice it.
    unique_citations = list(unique_citations.items())
```

Now that we have a dict (an OrderedDict), duplicate keys are implicitly removed. Only unique values are retained (of course, we lose some of the duplicate details, but finding

unique annotation+details combination doesn't reduce the no. of unique citations enough, it doesn't make sense to do it that way).
Making a list out of the items of the Ordered dict results in [(annotation, details), (annotation, details),…], just like we want for the unique citations.

Now, for each of the unique citations (if no. of unique citations is less than num_rows) or for 3 * num_rows citations from the unique citations, there is a call to search_sentences. This is used to retrieve all the citing papers containing the citation in question.

```
if num_rows * 3 < num_unique_citations:
        unique_citations = unique_citations[:num_rows*3]
    final_results = []
    for annotation, details in unique_citations:
        reslist = search_sentences(annotation, num_rows)
        if reslist == []:
            # If the annotation was not found, go to next annotation
            continue
        res = reslist[0]
```

If no results are returned (it's possible that the annotation from the refs files are not present in any of the papers text files, i.e. in the papers index), just move on to the next annotation, as shown below.

```
if res == []:
            # If the annotation was not found, go to next annotation
            continue.
```

All the values returned by the call to search_sentences are added to the result list as well.

```
        for entry in res:
            # res is a list of lists -> entry is a list: extract values from entry
            intermediate_result = []
            intermediate_result.append(annotation)
            intermediate_result.append(details)
            # Values of 1 result: title, authors etc.
            for value in entry:
                intermediate_result.append(value)
            # Append the current result to final_results. final_result contains full lists.
            final_results.append(intermediate_result)
```

When multiple sentences in a paper contain the same citation, they need to be grouped together under the same result. This is done in the group_sentences_together function.

```
def group_sentences_together(results):
    """ Takes a list of lists of results which may include multiple
sentences from the same CITING paper, and groups them
    together in a list. The final list of lists which is returned will have
fewer or equal results as the input list."""
    # Convert the list of lists into a dataframe, replace missing values (Nones are converted
into NaNs when a dataframe is created)
    df=pd.DataFrame(results, columns=['annotation', 'details', 'sentence',
'arxiv_identifier', 'title', 'authors', 'arxiv_url', 'published_date',
'dblp_url'])
    df['dblp_url'].fillna('dummy value', inplace=True)
    df_grouped = pd.DataFrame(df.groupby(['arxiv_identifier', 'title',
'authors', 'arxiv_url',
                                          'published_date', 'dblp_url',
'annotation', 'details'])['sentence'].apply(list)).reset_index()
```

```
    # Reorder the columns
    cols = ['annotation', 'details', 'sentence', 'arxiv_identifier',
'title', 'authors', 'arxiv_url', 'published_date', 'dblp_url']
    df_grouped = df_grouped[cols]
    grouped_list = df_grouped.values.tolist()
    for result in grouped_list:
        result[8] = None if result[8] == 'dummy value' else result[8]
    return grouped_list
```

After grouping, control returns to the seach_references function, which only keeps the number of rows asked for by the user and sorts the data by date.

```
    final_results = final_results[:num_rows]
    final_results.sort(key=lambda x: x[7].split(';')[0], reverse=True)
```

This is now sent to the views function which has the task of providing a mechanism by which the annotation in each of the sentences in the result can be highlighted in the template.
This is done by going through the list of lists, and using regular expressions to get the start and end index of the annotation in the respective sentence, the start of the sentence and the end index before the annotation, and the index after the annotation. This is shown in the code below.

```
def change_dateformat_addoffsets_citation(results):
    """ Changes the date format for all the results from yyyy-mm-dd into
Month dd, yyyy; """
    for result in results:
                # Foll. list will be of the form [[sentence1, annotation_index,
before_annotation_index, after_annotation_index], [sentence2,...],...]]
        sentence_with_annotations = []
        for sentence in result[2]:
            # sublist will contain 1 sentence, and three sets of indeices
            sublist = []
            match = re.search(result[0], sentence)
            sublist.append(sentence)
            # Find indices of annotation in sentence (separated by :), indices of the
sentence before the annotation and
            # indices of the sentence after the annotation, both also separated by a colon.
            # This is used in the template, where {{sentence|slice:annotation_indices}} is
used to get the part to highlight the annotation.
            sublist.extend(["{}:{}".format(match.start(), match.end()),
"{}:{}".format(0, match.start()), "{}:".format(match.end())])
            sentence_with_annotations.append(sublist)
        result[2] = sentence_with_annotations
    return results
```

The annotation can then be highlighted in the template using the following code.

```
{% for annotation, details, sentence_list, arxiv_identifier, title,
authors, arxiv_url, published_date, dblp_url in results %}
… … … <Some more html here>
       {% for sentence, annotation_indices, before_annotation_indices,
after_annotation_indices in sentence_list %}
       <p> Hit {{forloop.counter}}:
{{sentence|slice:before_annotation_indices}}
       <span class="cyan accent-2 black-
text">{{sentence|slice:annotation_indices}} </span>
   {{sentence|slice:after_annotation_indices}} <br/> </p>
        {% endfor %}
{% endfor %}
… … … <Some more html here>
```