Project Report

Group name: CLZZ
Students: Anhua Chen (anhua@uchicago.edu)
Xiang Zhang (snzhang@uchicago.edu)
Xiuyuan Zhang (xiuyuanzhang@uchicago.edu)
David Liu (dliu5@uchicago.edu)
Github repository: https://github.com/zhangxiang0822/CS123_Project

1. Project Overview

This project explores the topic of innovation diffusion across industries using patent citation data. Through analyzing the dynamics of the patent citation behaviors across industries, we aim to find the diffusion patterns of patents within and across industries. To achieve this goal, we perform several large data merges leveraging the Google Cloud Platform for parallel computing; further, we use the Dijkstra algorithm to find the shortest paths between each patent citation pair and aggregated these pairwise computations to the industry level. For this specific project, we focus on patents granted in the U.S due to the magnitude of the citation data. This means that, for a citation pair where A cited B, both A and B were patents granted in the U.S.

2. Data Description

We use a total of three datasets: (1) NBER patent classification data, (2) U.S. patent citation data, and (3) NBER industry classification data. All three datasets are downloaded through the United States Patent and Trademark office website: http://www.patentsview.org/download/

(1) U.S. patent citation data (98,207,057 rows, 3.505 GB). The U.S. patent citation data provides information on citations of U.S. patents made by U.S. patents. Each row of the dataset contains nine columns: universally unique id, patent number, patent number for which the current patent cites, date of when the cited patent was granted, name of the cited record, kind code from WIPO, country where the cited patent was granted (always U.S.), category (who cited the patent), sequence (order in which this reference is cited by select patent).

uuid	patent_id	citation_id	date	name	kind	country	category	sequence
00000jd7thmiucpaol1hm1835	5354551	4875247	1989-10-01	Berg	NULL	US	NULL	11
00000l0ooxrvfv6jkenobhwis	D674253	D519335	2006-04-01	Ishii	S	US	cited by examiner	13
00001nlwuimui60vu3k1yzjqd	8683318	6642945	2003-11-01	Sharpe	B1	US	cited by examiner	6
…	…	…	…	…	…	…	…	…

Table. 1 U.S. patent citation sample data

(2) NBER patent classification data (5,105,937 rows, 110.807 MB). Each row of the NBER patent classification data contains category information for a single patent, including the universally unique identifier for the patent, the patent number, a NBER category id, and a NBER subcategory id.

(3) NBER industry subcategory data (37 rows, 871 bytes). This data contains information of 37 pairs of subcategory IDs and subcategory names. The set of subcategories is developed by the National Bureau of Economic Research (NBER) and widely used for economic research.

id	name
11	Agriculture, Food, and Textiles
12	Coating
13	Gas
…	…

Table. 2 NBER patent classification sample data

uuid	patent_id	category_id	subcategory_id
000114qfli99qqd9fsbxichy1	6243839	2	22
0001jsdl1xi7z84rzx9iwvdlh	4646100	2	21
0001qpsb0yts8daudtuf3mbm8	7712627	6	68
…	…	…	…

Table.3 NBER industry subcategory sample data

3. Data Merge & Preprocessing

Our target in this section of the project is to merge very large datasets. Since we would like to see the industry to industry citation pattern and shortest path in our project. We need to merge:

- NBER (National Bureau of Economics Research) industry classification dataset
- Patent-NBER industry category ID dataset
- US Patent Citation dataset

In the following order:

1. Merge patent-NBER industry category ID dataset with NBER industry classification dataset to create a dataset of (patent_id, industry) pairs (~6 million obs).
2. Merging the (patent_id, industry) pairs with us patent citation dataset (~98 million obs)

We will utilize the MapReduce to perform the merging.

This section is structured as the following: firstly, we will introduce the structure of our code in the two phases of merging. Then we will discuss issues and challenges when trying to move our runners from local to Google Cloud Services (GSC).

3.1 1st Phase Merging

Since we are taking multiple datasets as input, we add the following try-except structure in the mapper function. The idea behind is to utilize the differentiated data structure of two tables. For the purpose of reducer phase later on, we need to yield the output of mapper in the same structure for two tables.

Since the input of reducer share the same structure, all we need to do is firstly sort the input within each key group, and then “broadcast” the industry name onto patent data to create (patent_id, industry_name) pairs. However, the sorting process might be complicated if running on GCS given multiple-instance, which we will discuss in detail later.

```
try: # See if it is main table record
     if var-structure satisfy that of table 1
```

```
            table_id = 1
            yield key, values
        except ValueError:
            try:
                if var-structure satisfy that of table 1:
                      table_id = 2
                      yield key, values
            except ValueError:
                pass
```

We also add a feature of choosing between different type of merging in our code (See following).

```
        for vars in values:
            if (str(self.options.merger_type) == "left"):
                if table_id == 2:
                    tit = title
                else: yield patent_id, tit

            elif (str(self.options.merger_type) == "outer"):
                if table_id == 2:
                    tit = title
                yield patent_id, tit

            elif (str(self.options.merger_type) == "inner"):
                if table_id == 2:
                    tit = title
                elif tit: yield patent_id, tit
```

In order to control for merging-type in command line, we also include a pass-through arguments configuration in the code (See the following).

```
    def configure_args(self):
        super(NBER_merge, self).configure_args()
        self.add_passthru_arg('--merger_type', default = "left_join",
            help = "Type of merging")
```

3.2 2nd Phase Merging

Compared with the first phase merging, 2nd phase is a little bit more complicated since we need to merge industry name with 2 patent_id in the uscitationpatent dataset. The general idea, however, remain the same. For each observation, we will yield 2, rather than 1, (key, value) pairs. (See code below)

```
yield key, (tit, "")    # citing obs
yield key, ("", tit)    # cited obs
```

In the reducer-phase, we just utilize a two-step reducing:

1. We "broadcast" industry name onto both citing and cited patent (generated from mapper)

```
Method broadcast(self, key, values):
        tit = None
        sort values
           if table_id == 2:
               tit = title_citing
           elif citing_order == 1:
                    yield key, (tit, "")    # citing obs
           else:
                    yield key, ("", tit)    # cited obs
```

2. Then we "aggregate" the citing observation with cited into one observation (please see pseudo code below).

```
    Method: aggregator(self, key, values):
        initialize value to be aggregated
        sort values

        for var_list in values:
                if citing_order == 2:
                    store value of cited obs
                else: yield final key-value pair
```

We then just use the step function to knit broadcast with aggregator.

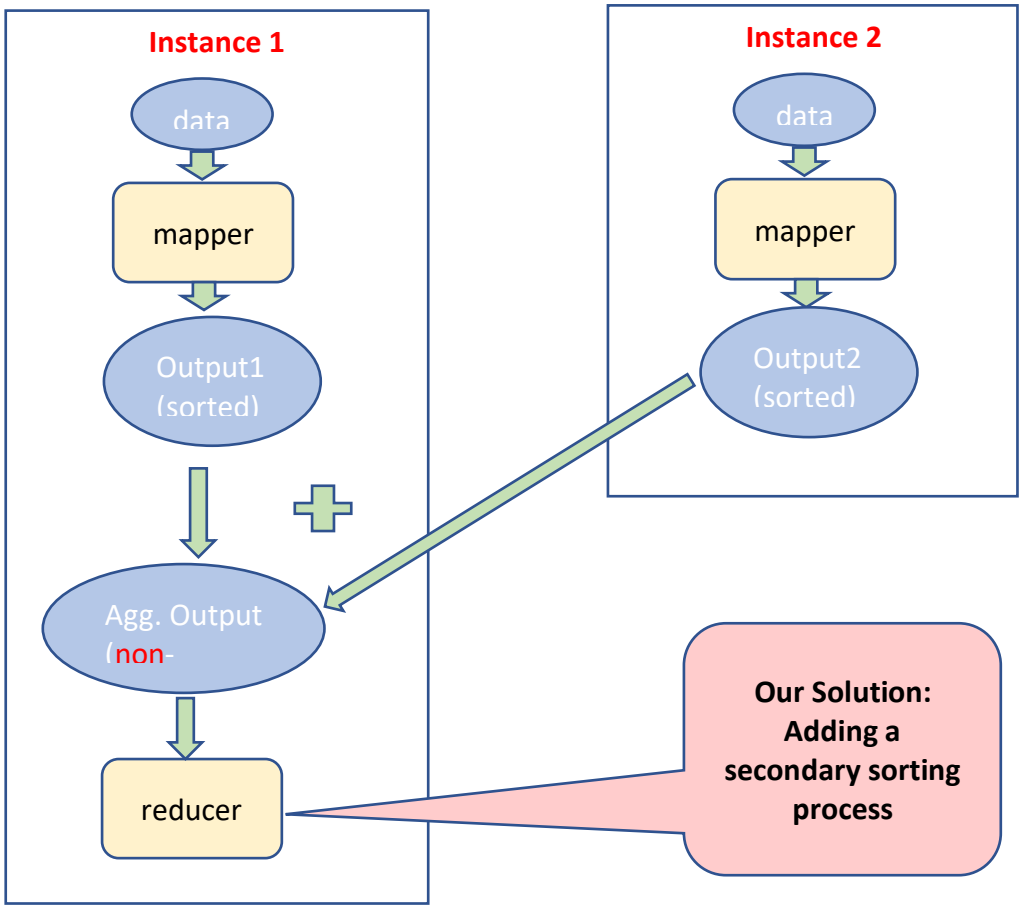
```
    def steps(self):
        return [MRStep(mapper=self.mapper,
                    reducer=self.broadcast),
```

```
            MRStep(reducer=self.aggregator)]
```

3.3 From Local to GCS

However, one of the challenges we run into is that merging results generated from local runner is different from that on GCS. We run the following series of "controlled-experiments" trying to debug the issue:

1. Altering the number of instances launched on GCS
2. Altering the number of reducer tasks on GCS
3. Setting the sorting process in reducer



We figured out the issue is due to the multi-instances launched on GCS. When running locally, the output from mapper will be aggregated and sorted before streamed into reducer. However, this is limited to the 1-instance case. In contrast, when running on GCS Dataproc, at least two instances have to be used. Although the mapper output for each key is sorted on each of the instances, when mapper outputs are aggregated and streamed to reducer, the "secondary-sorting" is not performed by default. Therefore, our merging code won't work on GCS.

We then prepare two solutions:

1. Adding a sorting process in the reducer method

```
if (str(self.options.runner_type) == "GCS" or
            str(self.options.runner_type) == "cluster")
            values = sorted(values,
                key=lambda x: x[0], reverse = True)
```

2. Setting the secondary sorting macro of MRJob to True

```
MRJob.SORT_VALUES = True
```

We go for the first option since it helps us to better understand and demonstrate the procedure on GCS[1].

4. Finding the Shortest Path

To explore how the network of innovation diffusion behaves both within and across industries, we first compute at the connections between all pairs of patents in the dataset through their citation records. We later use these results to aggregate from individual citations to industry level connections. To achieve our goal, we implement Dijkstra Shortest Algorithm to find the shortest path between nodes in a graph, where nodes represent the cited patents. In our implementation, we fix a single node as the "source" node and find the shortest paths from the source nodes to all other nodes in the graph. Further, to analyze the network structure of the graph, we iterate over all nodes, set them as source nodes, and find the shortest path between every possible node pairs.

We split this task into two steps: (1) design a Dijkstra structure for a sample data that contains only a part of our target dataset, and (2) restructure the code from (1) to run on Google Cloud for the entire dataset.

4.1 Small-data Dijkstra

In the small-data world, the Dijkstra algorithm works as follows (Gass and Fu, 2013)

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances through the current node. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one.

[1] This is the reason why we need to add a "runner_type " in argument configuration

4. When we are done considering all of the unvisited neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

In our implementation, we construct a data structure graph with two attributes:

1. *Node:* A list of node IDs.
2. *Edge:* A dictionary of list. The key of this dictionary is the node ID, and the value is a list storing ID of all of its neighbors.

The pseudo code is shown below

```
1  function Dijkstra(Graph, source):
2
3      create unvisited set Q (Now Empty)
4
5      for each vertex v in Graph:
6          dist[v] <- INFINITY
7
8          add v to Q
9
10     dist[source] ← 0
11
12     while Q is not empty:
13         u <- vertex in Q with min dist[u]
14
15         for each neighbor v of u and v in Q:
16             alt <- dist[u] + 1 //In our setting, distance are always 1
17             if alt < dist[v]:
18                 dist[v] <- alt
19
20         remove u from Q
21     return dist[], prev[]
```

4.2 Large-Data Dijkstra

In the large-data world, we want to parallelize the Dijkstra algorithm. In fact, we implement a breadth-first search.

Data representation: In large-data world, we slightly modify the graph structure. We use a single line to represent information of a node. A single line is of the following format with four parts:

ID|ID1, ID2, ID3, .. IDn|Color|Distance

The first part represents the node ID; the second part, separated by commas, stores IDs of its neighbor nodes; the third part is an indicator of whether this node have been visited/to be visited and can take three values; the last part stores the distance.

The Pseudo code is shown below

```
1  class Mapper
2      method Map(NodeID nid, NodeInfo n):
3          dist <- n.distance
4
5          if n.color = 'gray':
5              for each NodeID mid in n.neighbors:
6                  m.dist <- dist + 1
7                  m.color = 'gray'
8                  yield (NodeID mid, NodeInfo m)
9          n.color = 'black'
10         yield (NodeID nid, NodeInfo n)
11
12 class reducer
13     method Reduce(NodeID nid, NodeInfo n):
14         dist_min <- infinity
15         M <- []
16         Color <- 'white'
17         for each m in n.neighbors:
18             if m.distance < dist_min:
19                 dist_min <- m.distance
20             if m.neighbors != []:
21                 M.extend(m.neighbors)
22             if m.color = 'black':
23                 color = 'black'
24             if m.color = 'gray' and color = 'white':
25                 color = 'gray'
26
27         update NodeInfo n using dist_min, M, Color
```

```
28          yield (NodeID nid, NodeInfo n)
```

Compared with the Dijkstra algorithm implemented using small-data, the breadth-first search method implemented on paralleled machines is much less efficient. Recall that in Dijkstra algorithm, we identify the node with shortest distance to the current node and pursue only that edge. However, in the paralleled case, we explore all paths (iterate over all neighbors) parallelly and create lots of "wastes". We will further discuss implementation-wise issues in the following section.

4.3 Challenges for a code transformation from small to large data worlds

As we have discussed before, compared with the Dijkstra algorithm implemented using small-data, the breadth-first search method implemented on parallelized machines is much less efficient. When trying to implement it on Google Could, we also face more implementation issues. One issue is, when running a BFS, we are implementing map-reduce process iteratively, where the output of last reducer will be the input of next mapper. Here, a very natural issue arises. That is, how should we set up the number of iterations, or put it differently, how could the process stop automatically when all nodes are iterated over.

Our current solution is to set a "steps" method in our mrjob class. If we want to have a full list of nodes reachable within *n* steps, we set up the "steps" method as

```
def steps(self):
        return [MRStep(mapper=self.mapper,
                       combiner=None,
                       reducer=self.reducer)
                ] * int(self.options.iteration)
```

where the *n* parameter mentioned above is passed to the mrjob class using mrjob's `configure_args` method:

```
self.add_passthru_arg('--iteration', default = "3",
                      help = "Number of Map-reduce iterations")
```

We also have another issue about how to handle our inputs to the parallelized BFS. Conceptually, the program needs to know our source node Traditionally, the input file will have a line whose color is marked as "gray", and distance is set to be 0. Thus, when read a line with color "gray" and distance 0, the program would know that the node in this line is the source node. However, in big-data world (especially when we would iteratively setting up every node as our source node), this method cannot work, because it requests us to modify our input file over and over. That

being said, if we have 6 million patents, the input file which is produced using Google Cloud would have to be modified 6 million times, producing a huge waste of computation resources.

Our solution is the following: we try not to modify the input file itself, but implicitly pass the source node ID to the mrjob. At the same time, all distance in the input file is set to infinite (sys.maxsize) and all colors in the input file are set to "white". Thus, when the program encounters one node whose color is "white" and id the source node ID, we change the color of it to "gray" (which means we'll process it right now) and the distance to 0. In this way, we're able to create only one input file but still able to run BFS smoothly. Again, we use mrjob's `configure_args` method to pass parameters.

```
self.add_passthru_arg('--start_point', default = "-1",
help = "Starting point")
```

5. Results

5.1 Citation Pattern

One of our goal for this project is to examine the citation behaviors of patents across industries (subcategories). Using results obtained from the merged datasets, we find the aggregate count of each citing industry – cited industry pairs. In order to focus on the cross-industry citation occurrences, we exclude the cases where the citing industry is the same as the cited industry (which is the most common case, and its appearance on the graph makes the other occurrences less distinguishable).

Below, we plot the cross-industry citations for five time periods, from 1976s to 2016s. The time period included is the period in which the cited patent was created. While the most common case is patents within an industry citing other patents in the same industry, there are also an overwhelmingly amount of cross-industry citations. Keeping in mind the temporal changes for citation patterns cross industries, there are two changes that stand out the most. The first one is the change of the asymmetrical citation relationship between the Drugs industry and the Organic Compounds industry. While there used to be patents from the Drugs citing patents from the Organic Compounds industry, there has been a significant decrease of citations from the Drugs industry to Organic Compounds industry in the past fifty years. During the same period of time, the Organic Compounds industry's citation of the Drugs industry remained consistent. This pattern of change suggests a possible divergence of these two industries; moreover, there might be a potentially more symmetrical interaction of these two industries.

In contrast, a cluster of technology-related industries become more and more interactive through their citations during the past fifty years. These industries include: Computer Hardware & Software, Computer Peripherals, Electronic Business Methods and Softwares, and Information Storage industries. The mutual citations among these industries grew over decades, becoming the most cited cross-industry cluster in the plot.

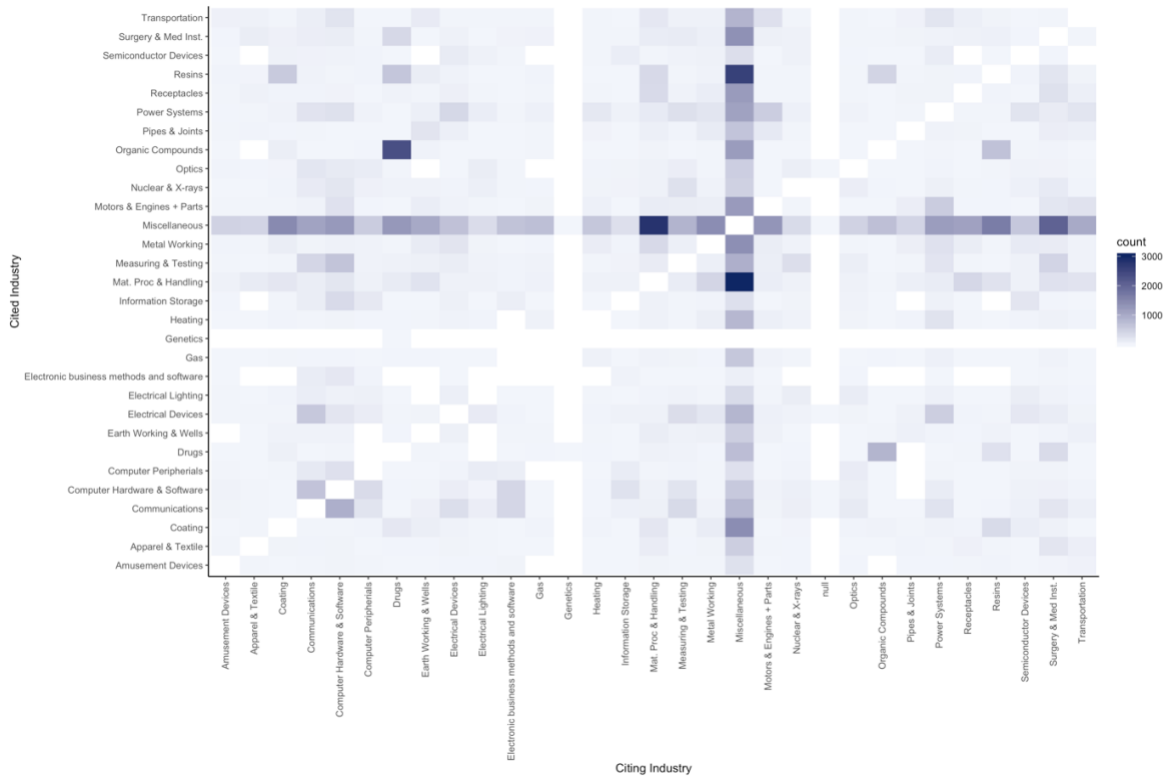


Figure 1. Cross-industry citations during 1976s - 1980s

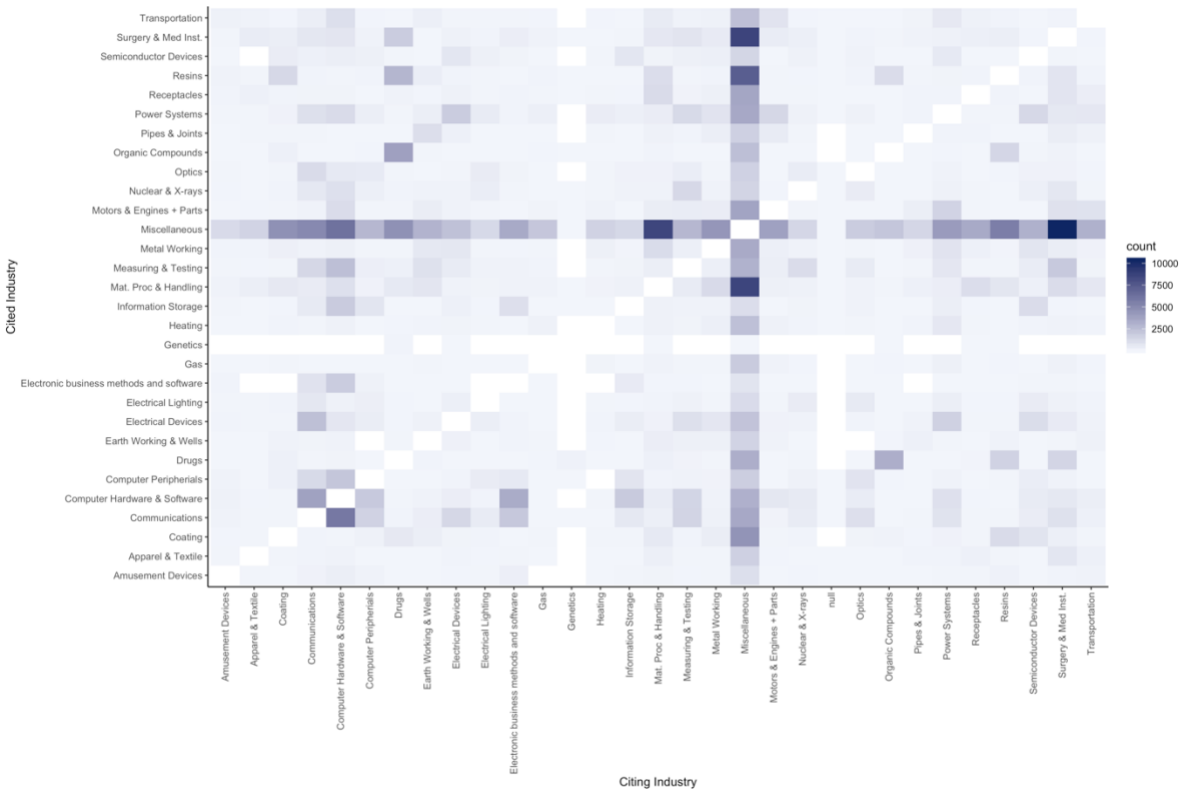


Figure 2. Cross-industry citations during 1981s - 1990s

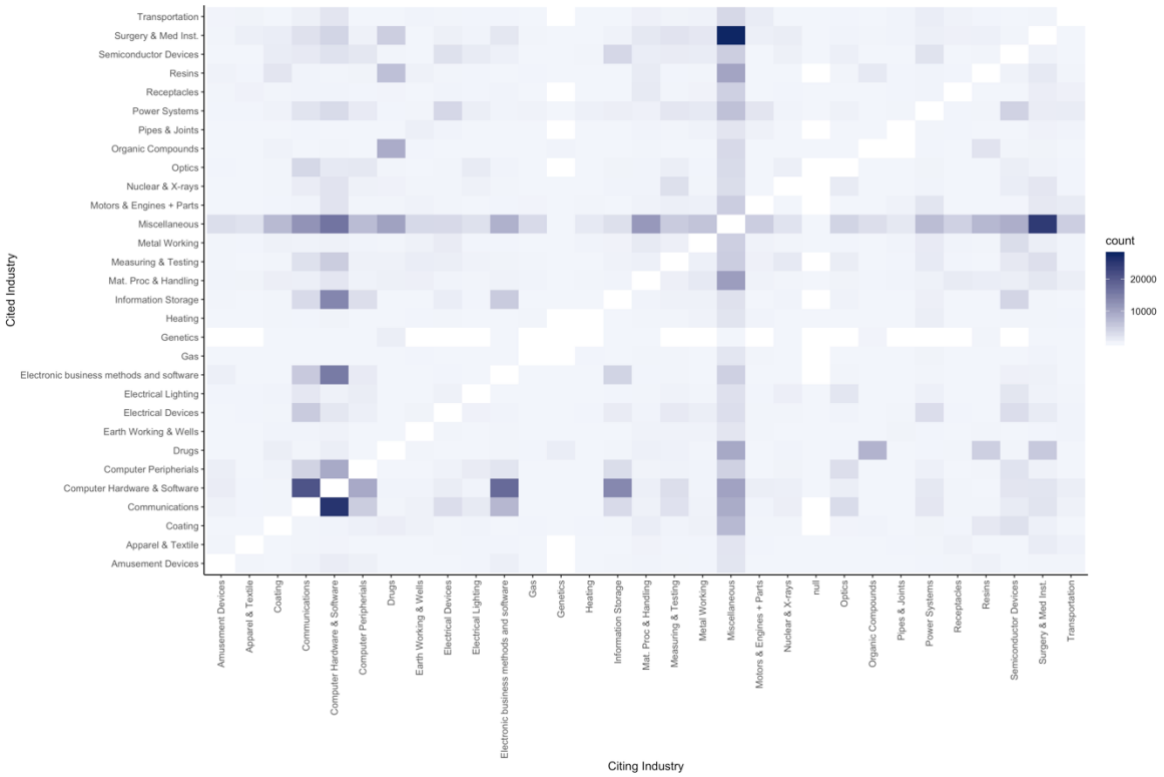


Figure 3. Cross-industry citations during 1991s - 2000s

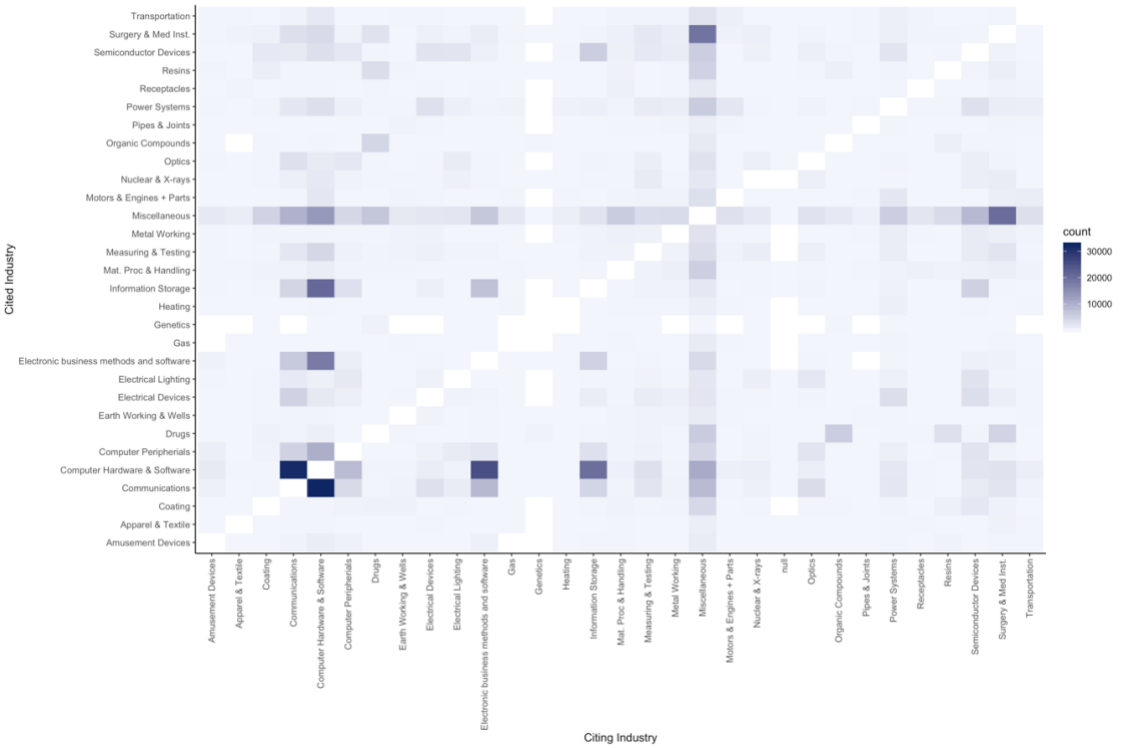


Figure 4. Cross-industry citations during 2001s – 2010s

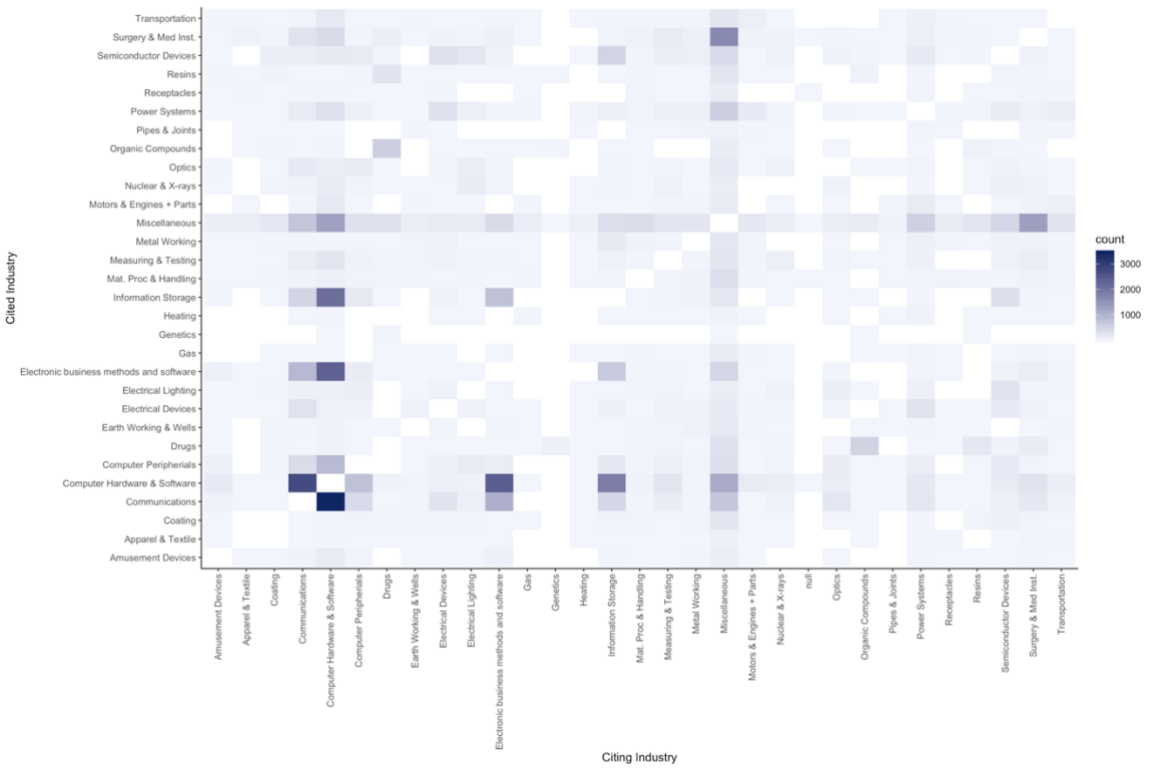


Figure 5. Cross-industry citations during 2011s - 2016s

5.2 Shortest Path from Single Node

Due to computational reason, we are not able to construct such network structure for all nodes, or even a small proportion of nodes (say, 10%, which is 600 thousand nodes). The arguments are as follows.

Conceptually, the time complexity of a single iteration of parallel Dijkstra is *O(m+n)*, where *m* stands for the number of edges in a graph structure, and *n* is the number of vertices. Since we want to find all nodes within *d* distance, the time complexity would be *O(m+n)* times $O(b^{d+1})$, where b is the branching factor of the graph. In our dataset, the total number of patent is 6 million and the total number of citation is 100 million, then we have a branching factor of about 16. In this case, if we want to produce the results for our entire nodes set, we will need $n \times O(m+n) \times O(b^{d+1})$ time. We have seen that a process with *O(nlog(n))* time complexity (our merging process) would take us four hours of work on Google Cloud Platform. When n equals to 6 million, we're conceptually not able to find the connected nodes for all nodes due to computation reason.

However, we still managed to implement our algorithm on Google Cloud for selected nodes. We set the allowed maximum distance to be 15 (which restricts $O(b^{d+1})$ to be less than 16^16). Figure 1 plots the network structure for nodes with a small number of connected nodes (with only 18 connected nodes), Figure 2 plots the network structure for nodes with a large number of connected nodes (the number of connected nodes reaches 4972).

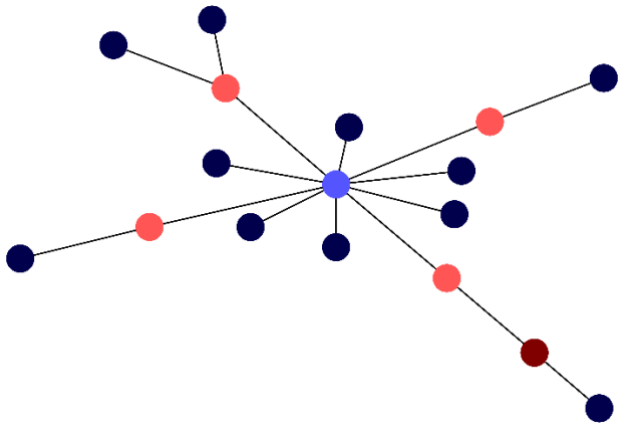


Figure 1. Node with Small Number of Reachable Nodes

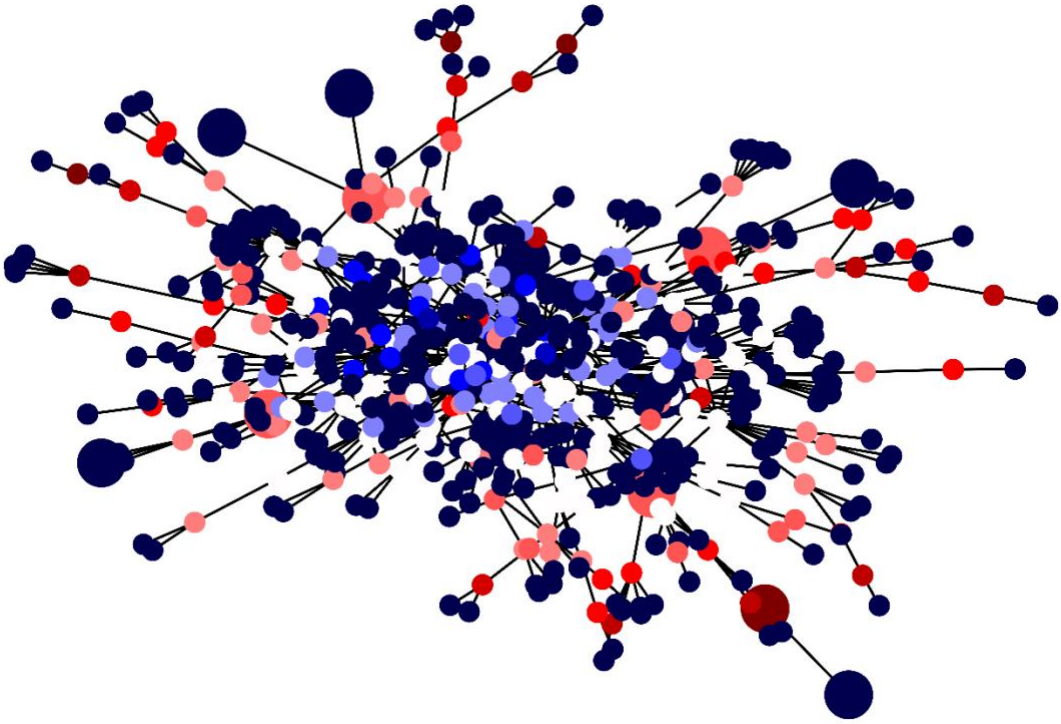


Figure 2. Node with Large Number of Reachable Nodes

Reference

Gass, Saul; Fu, Michael (2013). "Dijkstra's Algorithm". *Encyclopedia of Operations Research and Management Science*. Springer. **1**. doi:10.1007/978-1-4419-1153-7