Preliminaries: We have graphs saved in this Git repo under the three ‘data’ files. The paper being referenced is saved under ‘paper’. The three text files ‘assignments’, ‘ordered’ and ‘new’ are used by the algorithm, they don’t have any information pertinent to this write-up

**1. \*Explaining a problem and an algorithm to solve the problem.**

One of the first notable Agent allocation problems is known as The Assignment problem. Here, a set of N agents must complete M tasks. Each agent has a ‘time-cost profile’ over the tasks. The goal of this problem is to assign each agent to a task such that the total time to perform the tasks is minimized. This can be thought of in an Agent-Resource setting by setting the cost for each task as a ‘preference’ for a resource.

One of the first algorithms used to solve this problem is known as the Hungarian Algorithm, which can be solved in O(n3) time.

The problem we wish to solve is similar to The Assignment Problem, but there are a few slight differences. Again, we have N agents and M resources. Instead of a uniform preference profile over the resources, each agent has ‘indifference classes’, where in each indifference class each resource is viewed as equally desirable. The algorithm that we study works by “iteratively running lotteries for objects with lower demand (smaller equivalence classes), removing the assigned agents from equivalence classes, and then choosing the next smallest equivalence class (breaking ties at random),” (page 9).

**2. \*Implementing an algorithm.**

The algorithm was coded in Python, as we suspected the JRE would slow down the algorithm significantly. We also used Python to create graphs and data from the algorithm’s runs. For part 10, we used tables as well as these graphs to analyze the data.

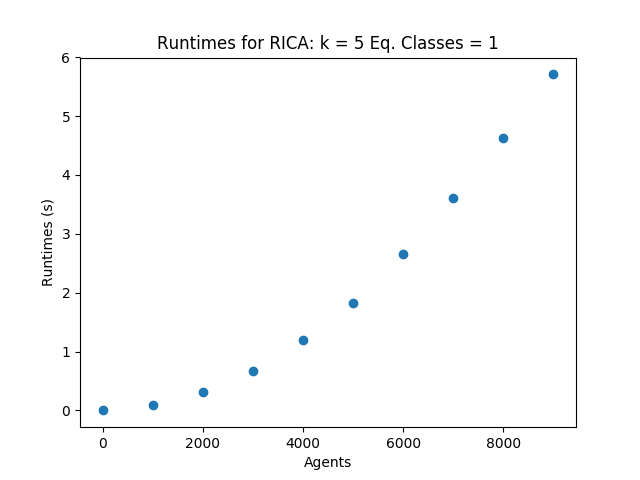
**3. \*Testing the implementation on a wide range of inputs, for correctness.**

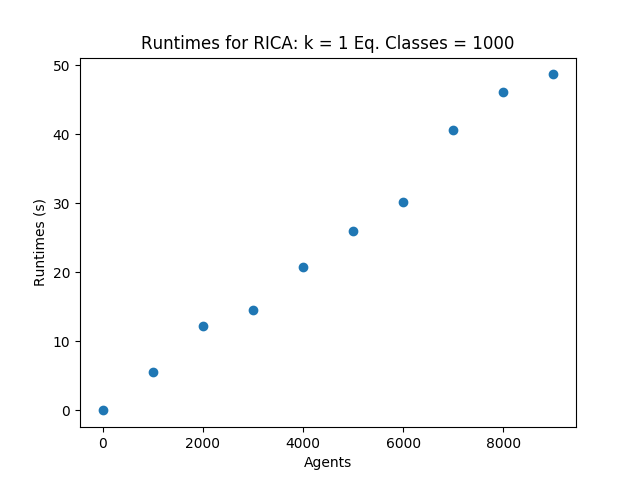
We tested the algorithm on sample sizes ranging from 100 – 10,000 agents, with equal number of agents and resources for simplicity. We could have varied the ratio of resources, but we already had many variables to deal with in regard to running time. A brief list of these is:

1. number of agents
2. number of resources
3. number of equivalence classes
4. number of resources in each equivalence class
5. ordering of resources

**4. Experimenting with the implementation to tabulate and graph the running time.**

The paper asserts in Theorem 5 that the overall running time for RICA is, in the worst case, O(n\*m + m2logm) (page 11). However, the worst case assumes that the number of indifference classes is equal to the number of resources (l = m). Due to limitations with the graphing package, we were only able to graph for a constant number of equivalence classes over any set of agents and resources. We were, however, able to show a linearithmic growth in the following two graphs over a random allocation of resources. Note that k is referring to the number of resources in each equivalence class



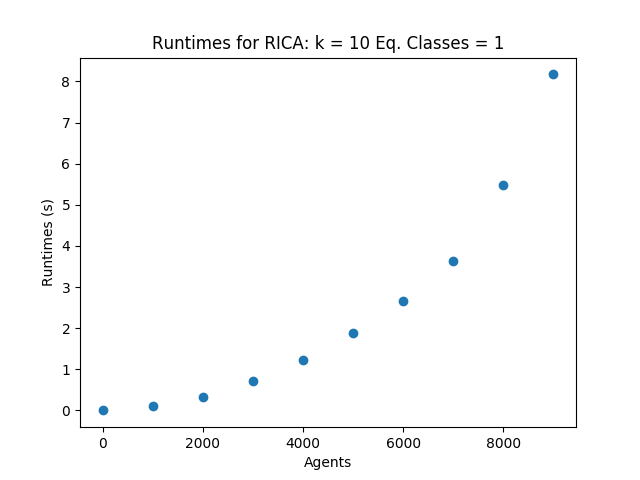


Note that as the number of equivalence classes increases, so does the runtime. Simply by changing the number of equivalence classes from a value of 1 to 1,000, the runtime looks very different. The runtime for 10,000 agents increased from 5.94 seconds to 48.20 seconds. This marked increase is due to the algorithm having to resolve 1,000 equivalence classes rather than just 1. The runtime for resolving an equivalence class is at most O(n2) as given by Theorem 3 (page 8).

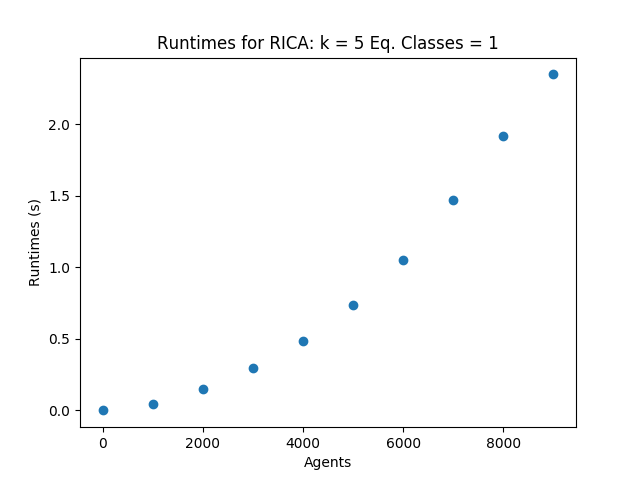
**5. Comparing running times on sorted, reverse-sorted, and random inputs.**

\*There are many more examples of the runtime of this algorithm on different sorted inputs in the folders. For sake of making this report readable, we only include graphs for one experimental setup, which will reflect the set as a whole\*

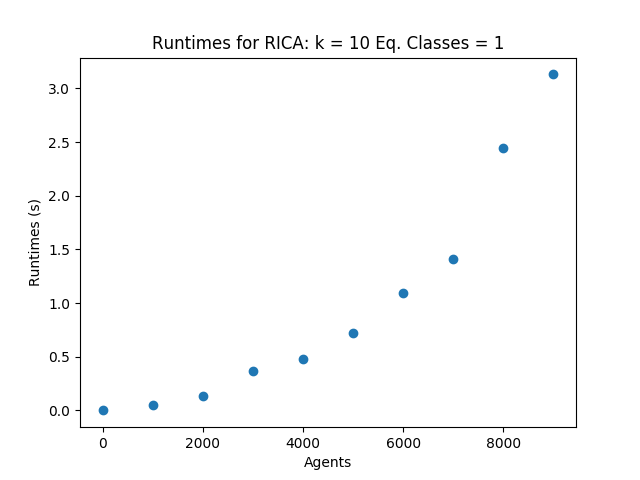
**Random Equivalence Class Allocations:**



**Sorted Equivalence Class Allocations:**



**Reverse Equivalence Class Allocations:**



These graphs give an interesting insight into how this algorithm functions in extreme cases. Let us consider how the data is processed in the ‘new’ folder by the algorithm. Given a line in the file, a set of resources is listed. Here, the line number in the file represents the agent:

(Agent Example) 116, 901, 312, 619, 138, 572, 556, 72

The number of equivalence classes is listed at the top of the file. For this example, suppose that the k-value is 2, and the number of equivalence classes is 4. Then, our above data will be partitioned into:

(Agent Example) 116, 901 | 312, 619 | 138, 572 | 556, 72

(note that this partitioning is a visual representation for what the algorithm is doing, not actually what the code executes)

Now, each equivalence class for the whole N agents is resolved in order, where the first equivalence class listed is seen as the highest priority, then the second class is resolved, etc.

When we sort our data, each line in the file is sorted. So the example agents preference profile after splitting into 4 equivalence classes becomes:

(Agent Sorted Example) 72, 116 | 138, 312 | 556, 572 | 612, 901

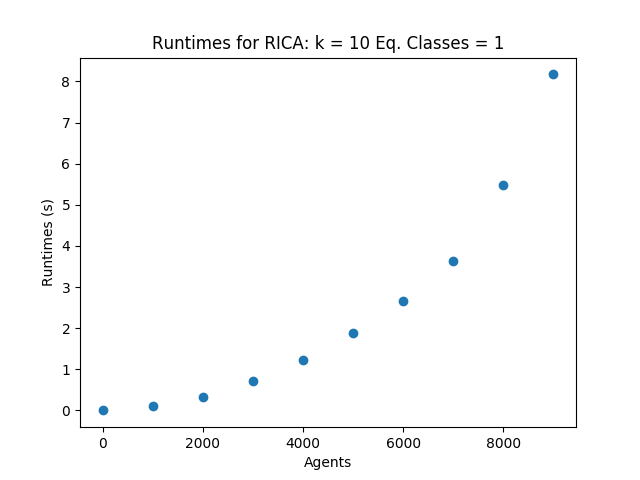
Intuitively there are more likely to be collisions of preferences here, as N agents will each have their preferences sorted in ascending (or indeed descending order). The algorithm deals with these collisions after all equivalence classes have been resolved. If there are any agents without resources, they are randomly assigned to any resource that has not been claimed. The authors of the paper claim that this preserves important measures of correctness for the algorithm (page 10).

As well as preserving correctness, this method for dealing with unassigned agents leads to a much quicker runtime once all equivalence classes have been resolved. As collisions are more likely in the sorted and reverse sorted cases, the runtimes for both should be smaller than the random input. This is indeed what the graphs above show. (Also note how this tie-resolving method effects the graphs for part 4).

**10. Analyzing the rate of growth of the algorithm on sorted, reverse-sorted, and random inputs.**

\*Here, we had the program output a list of running times in addition to the standard graphs. As with the previous section, we will only analyze 3 graphs that represent the whole data set, but there are more available in the Git files\*

**Random Equivalence Class Allocations:**



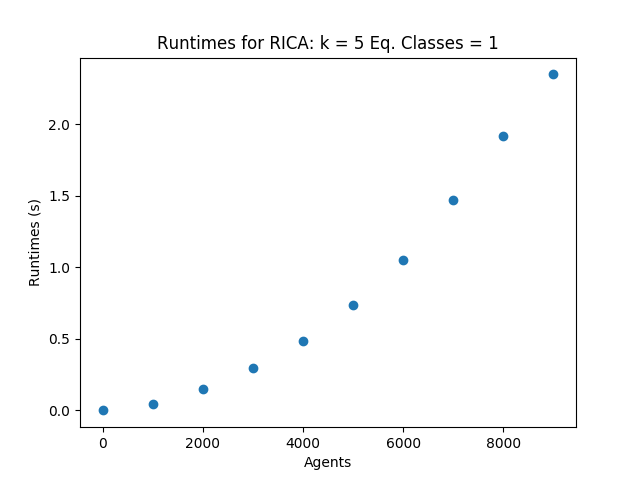
|  |  |
| --- | --- |
| Agents | Runtimes (s) |
| 0 | 0 |
| 1000 | 0.09 |
| 2000 | 0.32 |
| 3000 | 0.90 |
| 4000 | 1.55 |
| 5000 | 2.06 |
| 6000 | 2.87 |
| 7000 | 3.92 |
| 8000 | 6.11 |
| 9000 | 8.67 |

Looking at this data, we can look at the 1000, 2000, 4000, and 8000 data points to try and formulate an order of growth for the algorithm. Doing this yields the following:

|  |  |  |
| --- | --- | --- |
| Agents | Runtimes | Ratio |
| 1000 | .09 | 3.55 |
| 2000 | .32 | 4.82 |
| 4000 | 1.55 | 3.94 |
| 8000 | 6.11 | ---- |

It appears that the runtime follows a linear trend, however even with this number of agents, there is quite a bit of variation in the ratio between n and 2n agents. Assuming order N growth, our equation becomes runtime T(s) = 7.6x10-4 N.

**Sorted Equivalence Class Allocations:**



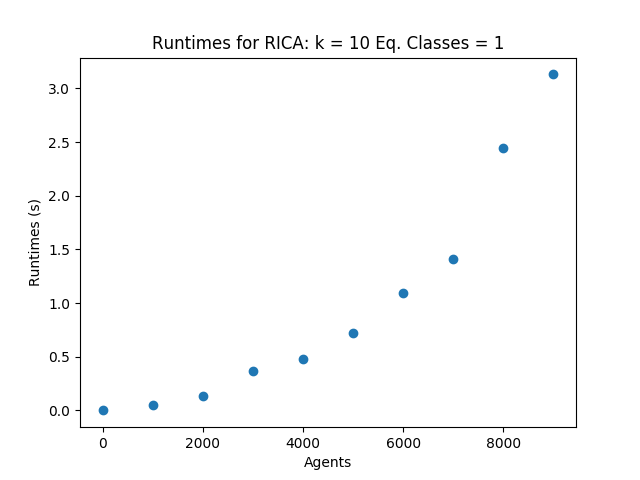
|  |  |
| --- | --- |
| Agents | Runtimes (s) |
| 0 | 0 |
| 1000 | 0.07 |
| 2000 | 0.39 |
| 3000 | 0.44 |
| 4000 | 0.52 |
| 5000 | 0.67 |
| 6000 | 1.13 |
| 7000 | 1.48 |
| 8000 | 2.00 |
| 9000 | 2.40 |

Using the same method as above:

|  |  |  |
| --- | --- | --- |
| Agents | Runtimes | Ratio |
| 1000 | .07 | 5.57 |
| 2000 | .39 | 1.33 |
| 4000 | 0.52 | 3.84 |
| 8000 | 2.00 | ---- |

These runtimes follow an even more variable ratio. One could extrapolate any result they wanted from this data. The analysis for sorted runtimes is inconclusive.

**Reverse Equivalence Class Allocations:**



|  |  |
| --- | --- |
| Agents | Runtimes (s) |
| 0 | 0 |
| 1000 | 0.09 |
| 2000 | 0.12 |
| 3000 | 0.47 |
| 4000 | 0.51 |
| 5000 | 1.30 |
| 6000 | 1.48 |
| 7000 | 1.51 |
| 8000 | 2.49 |
| 9000 | 3.25 |

Using the same methods as above:

|  |  |  |
| --- | --- | --- |
| Agents | Runtimes | Ratio |
| 1000 | .09 | 1.33 |
| 2000 | .12 | 4.24 |
| 4000 | 0.51 | 4.88 |
| 8000 | 2.49 | ---- |

Again, these runtimes follow a variable ratio. Visually, these runtimes and those of the sorted data look to follow a quadratic or linearithmic fit, but one cannot extrapolate that from the raw data.

Conclusion:

In conclusion, sorted and reverse sorted equivalence classes run about 2 seconds faster on average than random data. That said, random data has a much more ‘stable’ runtime, where one can extrapolate an equation for N agents. Larger numbers of equivalence classes greatly increase the running time of the algorithm, from 6 seconds to 50 seconds for 9,000 agents. Simply, there are too many variables to explore experimentally, however data has been extrapolated from the variable setups that we tested.