Algorithms for Intelligent Decision Making - Assignment 2

Group 28
Thalis Papakyriakou
Andrea Alfieri

March 2020

A Description of the algorithm

Our objective was to develop an algorithm for finding a collective schedule which minimises the sum of tardiness $(\Sigma - T)$ of all agents and all jobs compared to their preferred schedules, but which is also fair and is Condorcet-consistent.

In short, the algorithm we developed is the PTA Copeland's method as described in the paper "Collective Schedules: Scheduling Meets Computational Social Choice"[1], but with a slight modification at the end in order to enforce Condorcet consistency. We believe that this algorithm is one of the best choices for our objectives: The first part of the algorithm highly respects both the preferences of the agents and the job processing times, which in turn results in a low $(\Sigma - T)$. The second part minimally twitches the good resulting schedule in order to also make it Condorcet consistent. We proceed to explain the algorithm:

Part 1 As described in the paper, "For each job J_k we define the score of J_k as the number of jobs J_l such that at least $\frac{p_k * n}{p_k + p_l}$ agents put J_k before J_l in their preferred schedule (n = number of agents, p_k is the processing time of job with ID 'k'). The jobs are scheduled in the descending order of their scores"

Part 2 We traverse through the schedule created in part 1. If, for any adjacent pair of jobs, $J_k > J_l$ (which means job k is scheduled before job l), and it also holds that a majority of agents placed J_l before J_k in their respective preference lists, then swap their positions in the schedule. Repeat the above traversal and swaps until there are no more swaps to be made / the schedule is Condorcet-consistent.

The above algorithm has been proven to satisfy (up to a point) a lot of beneficial properties, like fairness, minimization of $(\Sigma - T)$, Pareto efficiency, Condorcet-consistency, PTA-Condorcet-consistency and runtime efficiency: all of the above are analysed in more detail later on.

Now that we have explained why we chose this algorithm, let us also explain why we dismissed some other options by process of elimination:

- The other algorithm proposed by the paper was "Iterative PTA Minimax", which focuses on defeat scores. Both the original version of this algorithm (Simpson-Kramer Min-Max Rule), and this adapted version, work poorly on the lower rankings when one candidate wins by an enormous margin, and are better suited for choosing the winner of the election than ranking the candidates[2].

- The Dodgson rule seemed like an interesting option, given that it focuses on the Condorset property and achieving it through hypothetical inversions/swaps on the preference lists[2]. Not being sure how well it would satisfy job processing times, however, we dismissed this option.

- Tideman's Ranked Pairs method sounded quite difficult to implement on its own[2], let alone adapt it to this scenario where processing times play a role. Thus, we also dismissed this option.

A.1 Implementation Details

The algorithm was implemented in Java.

The preference list of each agent is implemented as a hash map: The key is the ID of the job, while the value is the position where the agent places that job as a preference. This kind of implementation allows us to perform position lookups of time complexity O(1), when we have the ID of a job, a functionality that our algorithm relies upon.

[1] Krzysztof Rzadca Fanny Pascual and Piotr Skowron. "Collective Schedules: Scheduling Meets Computational Social Choice". In: *AAMAS 2018* (2018).

[2] Jonathan Levin and Barry Nalebuff. "An Introduction to Vote-Counting Schemes". In: *Journal of Economic Perspectives* 9.1 (1995), pp. 3–26.

A.2 Time and Space Complexity Analysis

In the pseudocode below, the time complexity has been added as a comment for every step that is not O(1). As we can see, the total time complexity of our algorithm is $O(j^2 * n)$, where 'j' is the number of jobs and 'n' is the number of agents.

The space complexity is $O(n * j)$, as we keep the preference list of each agent in memory for most of the duration of the algorithm.

A.3 Pseudocode

Algorithm 1 Modified PTA Copeland's Method

```
-                                                                   ▷ PART 1
for each pair of jobs i,j do                                        ▷ O(j^2)
    Number of agents who prefer i over j equals zero
    Number of agents who prefer j over i equals zero
    for each preferenceList of every agent do                       ▷ O(n) t
        if job i is higher in the preference list than job j then   ▷ This check is quickly performed
through two O(1) lookups in the hashMap/preferenceList
            Increment the number of agents who prefer i over j
        else
            Increment the number of agents who prefer j over i
        end if
        Calculate the threshold that job i needs to pass to get a point, defined as (p_i * n)/(p_i + p_j)
        (remember, n = number of agents, p_k is the processing time of job with ID 'k')
        if the number of agents who prefer i over j is equal to the threshold then
            Give a point to both jobs i and j
        else if the number of agents who prefer i over j is greater than the threshold then
            Give a point to job i
        else
            Give a point to job j
        end if
    end for
end for
Sort the jobs in descending order into array schedule, based on the total points they gathered.   ▷
O(j * log(j))
while schedule is not Condorcet-consistent do                       ▷ PART 2 is actually optimized in code
    for each adjacent pair of jobs i,j in the schedule, such that job i is before job j do   ▷ O(j)
        if the number of agents who prefer j over i is a majority then
            Swap jobs i and j in the schedule
        end if
    end for
end while
```

B Experimental Analysis

Number of Agents	500	100	500	1000	1500
Number of Jobs	100	500	500	1000	1500
Runtime - PART 1 (ms)	157	578	3785	36655	130527
Runtime - PART 2 (ms)	0	0	0	0	0
Total Runtime (ms)	157	578	3785	36655	130527
$(\Sigma - T)$ - after PART 1	51328195	258388271	1304480175	1824011781	956065502
$(\Sigma - T)$ - after PART 2	51806652	258662914	1308462454	1828722529	967440645
Gini - after PART 1	0.03381	0.01628	0.01473	0.01074	0.008924
Gini - after PART 2	0.03483	0.01629	0.01481	0.01076	0.008947
PTA-Condorcet violations - after PART 1	241	12575	6674	19661	36547
PTA-Condorcet violations - after PART 2	339	12714	7231	19736	36828
Pareto efficient	YES	YES	YES	YES	YES
Condorcet-consistent (ranking)	YES	YES	YES	YES	YES

Table 1: Table 1 displays the properties of the resulting schedule of our algorithm for some instances (column) as the number of agents and jobs grows. The job processing times are randomly chosen between 1 and 100. The main purpose here is to display how efficient the algorithm is in terms of runtime, but we thought that also posting the properties of each schedule could only be beneficial.
As we can see, the runtime significantly increases as the number of jobs and agents increases, but not exponentially. The number of jobs has a greater impact on the runtime than the number of agents, as we can see by the first two instances, which is something we predicted through the theoretical time complexity of our algorithm. Our algorithm can handle instances of more than 1000 jobs if given enough minutes.
We can also observe that transforming our resulting schedule from PART 1 into a Condorcet-consistent schedule takes minimal time (less than a millisecond), and thus costs nothing runtime-wise.
Finally, we can observe that transforming our resulting schedule from PART 1 into a Condorcet-consistent schedule harms every property (runtime, $(\Sigma - T)$, Gini, PTA-violations), with the sum-of-tardiness taking the greatest hit. However, it causes a new property to hold, which was a requirement from the assignment.
The tests were carried out on an Intel Core i7-8750H 2.2GHz CPU.

Let us now analyse the experimental results of the 3 benchmark sets we created: "Random", "ManyTies", "ProcessingTimesGap".

B.1 "Random" Benchmark Set

The "Random" benchmark set contains random preference lists and random job processing times, ranging from 50 to 100. The reasoning behind this set is that, being random, we could observe all different kinds of test cases within this set, and notice where our algorithm was lacking/superior, as well as where the Minizinc model was lacking/superior. Since we had to compare every schedule of our algorithm to the corresponding schedule the MiniZinc model produced, we kept the number of jobs ranging from 5 to 9, and the number of agents ranging from 30 to 34. Such parameters required the Minizinc model to have runtimes ranging from a few hundred milliseconds to a few dozens of seconds. We did not feel the need to stress the Minizinc model any more by testing higher parameters, because we have already presented the runtime capabilities of our algorithm, which far outperform the Minizinc model.

Page 6 presents the properties of the schedule our algorithm produced, and the schedule the Minizinc model produced, for all instances of the "Random" Benchmark set. All our algorithm's schedules are Pareto efficient, as defined in the paper[1], and Condorcet-consistent (ranking).

- Runtime: Not much to say here. Not even 1 millisecond was needed for our algorithm to produce a schedule with these parameters. Given that everything is random, we conclude that any instance of such low number of agents/jobs is easily solved.

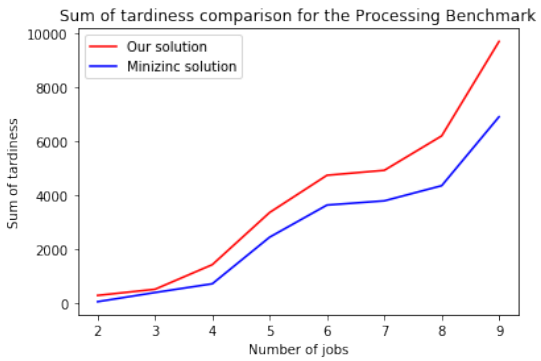
- $(\Sigma - T)$: Knowing that the Minizinc model produces the lowest sum-of-tardiness possible, we can see that our algorithm does a pretty good job of approximating it; we can observe that by taking a look at Figure 3a on page 5. The figure/graph presents the distance (%) of our solution to the optimal solution, which is defined by us as

  ($(\Sigma - T)$_ALGORITHM - $(\Sigma - T)$_MINIZINC) / $(\Sigma - T)$_MINIZINC * 100.

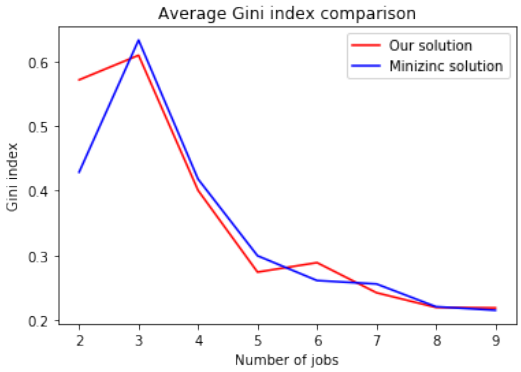
  That is, we subtract the optimal tardiness from our algorithm's tardiness, divide the result with the optimal tardiness and multiply by 100. As we can observe, the approximation is mostly within 10% of the optimal solution, with only 3 instances between 10 and 15 percent, and only 4 instances between 15 and 20 percent. This good result is mostly achieved by the first part of our algorithm, and the PTA-violations heuristic.

- Gini index: The Gini index of the Minizinc model is slightly better (lower) than ours. However, ours is really close, and it rarely goes above 0.2, indicating that our schedules are very fair.

- PTA-Condorcet violations percentage (%): Our algorithm, and especially the first part of it, is directly related with minimizing the PTA-Violations. That work is slightly harmed in part 2 of the algorithm, in order to achieve Condorcet-consistency. Nonetheless, we can see that our algorithm usually falls into less violations than the Minizinc model, with some exceptions. Additionally, the violations percentage varies a lot per instance, but rarely goes beyond 25%.



(a) Tardiness comparison for "ProcessingTimesGap" benchmark set with respect to the number of jobs



(b) Gini index comparison for "ProcessingTimesGap" benchmark set with respect to the number of jobs

Figure 1: "Processing Times Gap" Benchmark set analysis

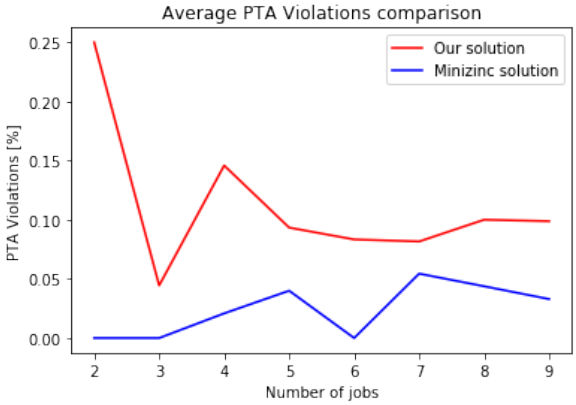
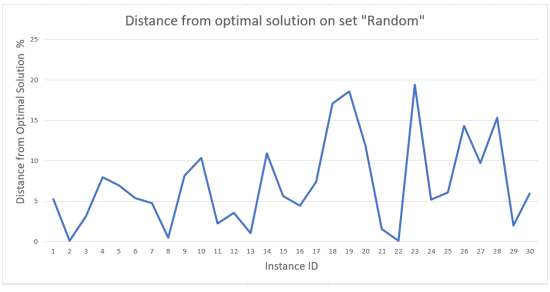
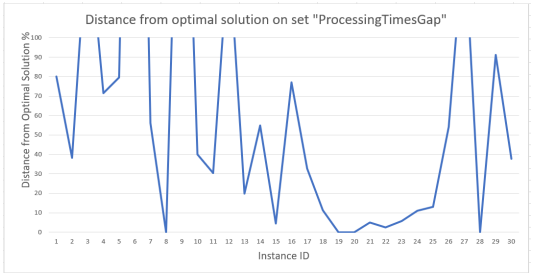


Figure 2: Comparison of the PTA Violations percentage on "ProcessingTimesGap" benchmark set



(a) Distance (%) of the schedule of our algorithm from the optimal $(\Sigma - T)$ solution on set "Random"



(b) Distance (%) of the schedule of our algorithm from the optimal $(\Sigma - T)$ solution on set "ProcessingTimesGap"

Figure 3: Distance (%) of the schedule of our algorithm from the optimal $(\Sigma - T)$ solution

	N:30 J:5		N:30 J:6		N:30 J:7		N:30 J:8		N:30 J:9		N:31 J:5	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	235	0	221	0	636	0	2347	1	20371	0	136
$(\Sigma - T)$	9364	8893	11395	11379	14842	14393	20164	18677	28691	26815	9094	8629
Gini	0.263	0.256	0.190	0.179	0.168	0.178	0.149	0.125	0.157	0.149	0.22	0.162
PTA viol%	8	24	0	5	4	12	18	12	14	14	16	24

	N:31 J:6		N:30 J:7		N:31 J:8		N:31 J:9		N:32 J:5		N:32 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	187	1	522	0	2025	0	36209	0	135	0	181
$(\Sigma - T)$	9477	9044	14145	14076	20691	19134	29411	26648	9973	9755	12291	11867
Gini	0.165	0.181	0.146	0.134	0.178	0.150	0.142	0.117	0.284	0.254	0.223	0.189
PTA viol%	5	16	8	8	12	9	32	19	8	16	5	16

	N:32 J:7		N:32 J:8		N:32 J:9		N:33 J:5		N:33 J:6		N:33 J:7	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	372	1	2212	0	23133	0	134	0	193	0	472
$(\Sigma - T)$	16905	16730	20741	18700	27386	25928	11184	10706	14661	13652	21444	18316
Gini	0.141	0.146	0.171	0.132	0.163	0.156	0.197	0.197	0.225	0.144	0.182	0.164
PTA viol%	0	12	15	12	14	17	8	40	16	38	24	8

	N:33 J:8		N:33 J:9		N:34 J:5		N:34 J:6		N:34 J:7		N:34 J:8	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	2421	0	25143	0	134	0	190	0	499	0	1491
$(\Sigma - T)$	23827	20093	36072	32240	8385	8257	13640	13620	19481	16319	21816	20742
Gini	0.154	0.141	0.139	0.171	0.243	0.253	0.239	0.246	0.175	0.149	0.164	0.159
PTA viol%	25	9	19	22	16	8	16	22	20	8	3	6

	N:34 J:9		N:35 J:5		N:35 J:6		N:35 J:7		N:35 J:8		N:35 J:9	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	1	11861	0	144	0	198	0	472	0	2390	0	17466
$(\Sigma - T)$	27499	25921	8352	7306	11775	10731	18317	15885	20810	20398	27120	25593
Gini	0.167	0.162	0.174	0.213	0.221	0.187	0.187	0.158	0.172	0.180	0.176	0.156
PTA viol%	4	7	16	0	22	5	36	12	6	6	7	9

Table 2: Displays the properties of the resulting schedules of both our algorithm (J) and the Minizinc model (M) for all the instances of the "Random" benchmark test. N:number of agents, J:number of jobs. Random processing times, between 50 and 100.

B.2 "Processing Times Gap" Benchmark Set

The "Processing Times Gap" benchmark set contains random preference lists. Half the jobs have processing times ranging from 1 to 15, and the other half ranging from 80 to 100. The reasoning behind this set is that we would get to see the impact of the extreme scenario of only short and long jobs in our schedules, given the properties we are trying to achieve. The number of agents and jobs is random, but, again, because of the runtime limitations of the Minizic model, we kept the number of jobs ranging from 2 to 9, and the number of agents ranging from 5 to 28.

Page 8 presents the properties of the schedule our algorithm produced, and the schedule the Minizinc model produced, for all instances of the "Processing Times Gap" Benchmark set. All our algorithm's schedules are Pareto efficient, as defined in the paper[1], and Condorcet-consistent (ranking).

- Runtime: Again, our algorithm outperformed the Minizinc model on runtime: while our solution took on average 0.3ms to terminate, the given model finished on average in 1.9s.

- $(\Sigma - T)$: As shown in Figure 1a on page 5, our model's tardiness is, on average 44.3% more than the Minizinc's solution. This approximation is far worse than the approximation we had achieved for the "Random" benchmark set. This bad approximation can also be seen in Figure 3b (see the sum-of-tardiness part of section B1 for a reminder of the formula used). The reason behind this "flaw" is the gap between the processing times of this benchmark test, and the fact that, while our algorithm is trying to turn the schedule Condorcet-consistent during part 2, it greatly harms the sum-of-tardiness, since the schedule is now extremely sensitive to changes because of the extreme processing times. Additionally, the very fact that the gap between processing times is so big, makes any schedule more susceptible to bad results.

- Gini index: As shown in Figure 1b on page 5, our model performed better than the Minizinc model on the majority of the tests. This, combined with the $(\Sigma - T)$ results, show that our algorithm prefers to distribute more tardiness to all agents, but in a "more fair" way, in order to penalise each agent less. This comes to back up our claim concerning our bad approximation of $(\Sigma - T)$: our algorithm sacrifices the sum-of-tardiness to achieve Condorcet-consistency and fairness, and this seems like a necessary tradeoff.

- PTA-Condorcet violations percentage (%): The number of PTA-violations for this benchmark test, for our algorithm, is the worst. We attribute this flaw to the fact that we are attempting to make the schedule Condorcet-consistent during part 2 of the algorithm. Although we haven't tested the number of PTA-Violations for these instances after the end of part 1 of the algorithm, we know that the Condorcet-consistency rule does not take in account the processing times of the jobs, thus, any swaps performed during step 2 would be quite negatively affecting the number of PTA-violations, and that seems to be the case. Additionally, like we mentioned before, these extreme instances in terms of processing time make any schedule more susceptible to bad results. These results are shown in Figure 2 on page 5.

	N:25 J:4		N:28 J:8		N:12 J:3		N:7 J:8		N:18 J:4		N:7 J:2	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	202	1	2362	0	128	0	828	0	138	0	129
$(\Sigma - T)$	2600	1444	10303	7453	682	264	2565	1496	1876	1045	285	52
Gini	0.413	0.380	0.241	0.234	0.643	0.724	0.146	0.312	0.306	0.390	0.571	0.428
PTA viol%	37	12	15	3	22	0	25	6	25	0	50	0

	N:18 J:5		N:5 J:3		N:9 J:4		N:28 J:9		N:24 J:6		N:9 J:4	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	145	0	138	0	133	0	16655	0	168	0	140
$(\Sigma - T)$	3417	2190	188	188	1858	492	13343	9521	4737	3633	970	395
Gini	0.297	0.335	0.574	0.574	0.357	0.465	0.210	0.168	0.288	0.260	0.450	0.411
PTA viol%	32	8	0	0	50	0	19	4	16	0	36	0

	N:20 J:8		N:22 J:9		N:10 J:4		N:23 J:8		N:21 J:7		N:10 J:8	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	1693	0	14599	0	136	0	1806	0	307	0	990
$(\Sigma - T)$	5726	4772	11421	7370	369	353	9295	5252	5582	4202	3091	2780
Gini	0.244	0.189	0.239	0.244	0.439	0.433	0.278	0.145	0.239	0.260	0.185	0.221
PTA viol%	20	9	29	4	0	12	34	12	20	8	3	12

	N:17 J:5		N:27 J:3		N:8 J:7		N:26 J:5		N:20 J:5		N:13 J:5	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	136	0	132	0	265	0	143	0	141	1	135
$(\Sigma - T)$	2103	2103	1064	1064	1972	1878	3680	3593	2636	2490	1989	1793
Gini	0.277	0.277	0.560	0.560	0.204	0.290	0.282	0.276	0.274	0.292	0.264	0.308
PTA viol%	8	8	0	0	0	12	0	8	8	8	8	8

	N:12 J:9		N:10 J:4		N:20 J:5		N:8 J:3		N:7 J:3		N:24 J:7	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	8741	0	128	0	151	0	138	0	135	0	387
$(\Sigma - T)$	4322	3822	872	556	6324	2494	238	238	386	202	7208	5288
Gini	0.205	0.231	0.433	0.426	0.246	0.305	0.634	0.634	0.633	0.670	0.282	0.215
PTA viol%	9	9	25	0	56	8	0	0	22	0	28	12

Table 3: Displays the properties of the resulting schedules of both our algorithm (J) and the Minizinc model (M) for all the instances of the "ProcessingTimesGap" benchmark test. N:number of agents, J:number of jobs. Processing times between 1-15 or between 80-100.

B.3 Many Ties Benchmark

The goal of this benchmark is to analyse how our algorithm and Minizinc model perform when the set of preferences presents many ties in the job scores obtained by the end of part 1. The tests were all generated with 6 jobs and a variable number of agents. The processing times were always random in each test instance. 10 tests were generated with 720 agents, where each of them had one possible permutation of the 6 jobs as his preference. Therefore, all 6 jobs had the same amount of preferences for each position in the schedule, thus making the processing times the only relevant factor for the voting rule. 10 tests were generated by always having the job IDs {1, 2, 3} and their permutations in the top 3 positions and the job IDs {4, 5, 6} in the last 3 positions. Therefore, $6 * 6 = 36$ agents were needed. This way, all the jobs belonging to the first set had the same score compared to each other, but they were consistently more preferred than the other 3 jobs. The same concept was then applied in order to generate 10 more tests where the possible permutations where within the sets {1, 2}, {3, 4} and {5, 6}. In this latest case, the number of agents needed to cover the permutations was $2 * 2 * 2 = 8$.

As expected,the Minizinc and our solution provided the exact same solutions over the entire set of tests. This means that with many ties our algorithm is able to provide the optimal solution, but in a very small fraction of the time Minizinc takes to execute.

	N:720 J:6		N:720 J:6		N:720 J:6		N:720 J:6		N:720 J:6		N:720 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	1	850	1	607	0	522	1	515	0	537	1	514
$(\Sigma - T)$	292128	292128	269040	269040	313104	313104	273024	273024	259536	259536	269496	269496
Gini	0.161	0.161	0.160	0.160	0.159	0.159	0.160	0.160	0.157	0.157	0.158	0.158
PTA viol%	0	0	0	0	0	0	0	0	0	0	0	0

	N:720 J:6		N:720 J:6		N:720 J:6		N:720 J:6		N:36 J:6		N:36 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	1	501	0	532	1	525	0	510	1	177	0	149
$(\Sigma - T)$	228432	228432	326832	326832	335808	335808	228264	228264	5742	5742	7374	7374
Gini	0.163	0.163	0.162	0.162	0.165	0.165	0.155	0.155	0.223	0.223	0.227	0.227
PTA viol%	0	0	0	0	0	0	0	0	0	0	0	0

	N:36 J:6		N:36 J:6		N:36 J:6		N:36 J:6		N:36 J:6		N:36 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	134	0	144	0	135	0	137	0	153	0	154
$(\Sigma - T)$	6684	6684	5808	5808	6300	6300	6090	6090	5916	5916	6870	6870
Gini	0.217	0.217	0.222	0.222	0.222	0.222	0.225	0.225	0.220	0.220	0.222	0.222
PTA viol%	0	0	0	0	0	0	0	0	0	0	0	0

	N:36 J:6		N:36 J:6		N:8 J:6		N:8 J:6		N:8 J:6		N:8 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	135	0	149	0	135	0	130	0	127	0	129
$(\Sigma - T)$	6096	6096	6612	6612	900	900	840	840	688	688	828	828
Gini	0.221	0.221	0.221	0.221	0.329	0.329	0.324	0.324	0.326	0.326	0.332	0.332
PTA viol%	0	0	0	0	0	0	0	0	0	0	0	0

	N:8 J:6		N:8 J:6		N:8 J:6		N:8 J:6		N:8 J:6		N:8 J:6	
	J	M	J	M	J	M	J	M	J	M	J	M
Runtime	0	127	0	126	0	128	0	127	0	125	0	125
$(\Sigma - T)$	864	864	864	864	928	928	824	824	840	840	700	700
Gini	0.324	0.324	0.324	0.324	0.320	0.320	0.322	0.322	0.335	0.335	0.316	0.316
PTA viol%	0	0	0	0	0	0	0	0	0	0	0	0

Table 4: Displays the properties of the resulting schedules of both our algorithm (J) and the Minizinc model (M) for all the instances of the "ManyTies" benchmark test. N:number of agents, J:number of jobs. Random processing times between 50 and 100.

C Running the program

Unizip the folder and traverse to the folder directory using the command line. Use the following command for compilation

javac votingRules/.java Main.java*

and the following command for running the program

java Main 10 10

where 10 10 are examples of the 2 arguments you need to give: the number of agents and the number of jobs. If all goes well, a random instance of that size will be created (random preference lists and random processing times between 50 and 100), and you should be able to see a similar output:

PREFERENCES: TestInstancenumAgents=10, numJobs=10
, processing times: [66, 71, 62, 86, 92, 73, 84, 93, 64, 67]
, preferences=[[3, 5, 9, 2, 7, 0, 4, 6, 1, 8]
, [0, 8, 2, 5, 1, 9, 4, 7, 3, 6]
, [5, 6, 3, 8, 2, 9, 0, 4, 1, 7]
, [0, 5, 2, 4, 8, 1, 3, 9, 6, 7]
, [9, 4, 8, 3, 1, 0, 6, 7, 2, 5]
, [4, 3, 7, 2, 5, 6, 1, 9, 8, 0]
, [7, 0, 6, 8, 3, 9, 2, 4, 5, 1]
, [8, 7, 2, 0, 6, 5, 4, 9, 3, 1]
, [2, 5, 3, 8, 4, 7, 0, 6, 9, 1]
, [3, 9, 5, 0, 6, 4, 8, 2, 7, 1]
]
Schedule: [8, 3, 2, 5, 0, 9, 4, 7, 6, 1]
OUR TOTAL RUNTIME: 40 ms
PTA violations % for our solution: 0.06
Sum of Tardiness for our solution: 9160
Tardiness per agent: [732, 792, 835, 1011, 1237, 1144, 1146, 830, 633, 800]
Pareto Efficient schedule: true
Gini Index for our solution: 0.11716157205240152

D Conclusion

The Minizinc model always produces a schedule with better sum-of-tardiness than our algorithm, which was completely expected since it always finds the schedule that minimizes it. However, our algorithm's sum-of-tardiness greatly approaches the optimal values.

Our algorithm usually produces a better Gini index than the Minizinc model, which would indicate our schedules are more fair towards every agent. That is most probably a result of making each schedule Condorcet-consistent, at the cost of PTA-Condorcet consistency and sum-of-tardiness.

As for the PTA-Condorcet violations, our algorithm generally achieved a smaller percentage of violations than the Minizinc model, with certain instances turning the tables. The violations percentage constantly stayed below 25 for non-extreme instances.

Where our algorithm really stands out is its runtime efficiency, which enables it to scale to instances of thousands of jobs, and the fact that it always produces a Condorcet consistent schedule, which was a requirement by the assignment.

In conclusion, it looks like our initial intuition about the algorithm was right: By first focusing on PTA-Condorcet consistency, we implicitly focus on the processing times and the sum-of-tardiness. Then, by only performing the necessary swaps, we keep most of the benefits of the intermediate schedule, while also making it Condorcet-consistent and boosting fairness. That makes our algorithm greatly satisfy all of the aforementioned properties, while being extremely good at runtime and fairness.

E Feedback

This second project of the course has been appreciated by both of us because it gave us the opportunity to discuss how to reach an optimal solution for a practical problem, which also allowed us to test the real performance of our algorithm.

Additionally, the whole process of searching for an algorithm to satisfy some voting properties has been really fun. The amount of parameters we had to evaluate our algorithm on where quite many, but although time-consuming, the process was interesting.

The only negative aspect of this experience has been the Minizinc model, which was taking minutes and hours to solve more complex inputs. A faster model would have allowed us to properly test and compare our solution with a bigger amount of jobs and agents.b