Project and Task Assignments Using Adapted Once Cut Suffices and Moving Knife with Epsilon Algorithms

Group members: Michael Sippo, Daniel Li, Mary Chou
Repo name: penguin

Summary

Our project involves investigating how the cake cutting algorithms we learned in class can be applied to project and task division in a work environment. A project involves many sub-tasks that come with time and monetary costs. Business and Information Technology (IT) teams may have different preferences about which tasks take priority and about how to split the tasks across several projects. Additionally, teams need to determine how to split the tasks among development groups and ensure everyone is getting a fair share of the work. Our aim is to analyze the distribution of work among projects and development groups utilizing modified One Cut Suffices (OCS) and Moving Knife (MK) cake cutting algorithms.

Description

Our project focuses on trying to replicate a real world division scenario of a set of tasks over a year long project and trying to divide the work for certain projects among development team members. Our team focused on using the Once Cut Suffices and Moving Knife algorithms based on the rules and strategies discussed in this semester's course.

We started with the basic framework of the algorithms and then made additional enhancements to better meet the real world scenario we chose to simulate. Enhancements to the algorithms included passing in an epsilon value to create a threshold by which to limit envy, and changing the number of platters of cake produced. We tried to recreate the strategies of epsilon as described in the course's lecture on Near Exact Division algorithms. The OCS algorithm was enhanced to accept a variable that would allow as many platters to be created by the user as is desired. Thus we named this modified algorithm N Cuts Suffice (NCS).

Our preference files were tailored to represent the priorities of business teams versus IT teams. We also introduced a method of using a third preference file to represent the actual cost of each task. Our project attempted to accomplish two goals. The first was to take a year long project and use NCS to divide the work accross *N* projects that fell within a specified budget for a year. The results were then compared with the actual real world deviation to see how well the algorithm worked and how closely the results matched the real world scenario. The second goal was to take a smaller project, and divide the work among members of an IT team based on their preference using the Moving Knives with Epsilon (MKE) algorithm. The results are compared to actual real world assignments to see if the division of work would be fair and envy free to each individual member.

Approach to Modeling and Coding

Our first approach was to create the OCS algorithm using the rules from the course's lecture. We built the initial code based off Professor Cytron's Cut and Choose demo program. Our OCS algorithm took in 5 parameters: two preference files, a value dubbed *m*, a *L* value, and a *R* value. The *L* and *R* values are used to represent the values of the two platters that OCS creates. We took in both values so that we could scale their sum to 1. Our algorithm uses the *m* value to determine the number of initial pieces the cake is cut into. The *m* pieces are then sorted from least to great value as stated in the class notes. Once the pieces were sorted, we started creating a left platter by popping off items from the list until the *L* value was met or exceeded. If the value matched *L*, the left platter was completed and the remaining pieces were put onto the right platter. Otherwise, the last item put on the left platter was removed and stored as the *PieceToCut*. Using the equation *L - (CurrentLeftPlatter - PieceToCut)*, the algorithm determines the *a* value (as defined in the class notes) that must be found in order to satisfy the *L* value. We used a default resolution of .01 to move across the *PieceToCut* until we found the cut that would equal or be just over the value of *a*. (This resolution would become the basis to include a new input variable *epsilon* to allow a user to specify how fine the resolution would be to meet the goals of a Near Exact Division algorithm.) Once the cut position was determined, the *PieceToCut* is cut into two pieces

where the piece equivalent to the value of *a* was put on the left platter and the remainder of the piece (called *b)* was put on the right platter. The remaining *m* pieces are appended to the right platter and both platters are then returned. This algorithm was stored in the repository folder *alg 2*.

Once we had OCS running, we began to modify it to create the NCS algorithm. This algorithm now takes in 7 parameters: 3 preference files, a value *m*, a value *epsilon*, an integer called *order*, and a list of values that are stored in a list called *PList*. The value *m* is used in the same manner as in the original OCS. The value *epsilon* allows the user to determine the granularity for cutting the *PieceToCut* in the algorithm. This value is used to set the resolution, which was originally set to a default of .01 in OCS. The number of decimal positions in epsilon is also used as the number of decimals to round to for all piece points in the algorithm. The 3 preference files and the value *order* were used together in creating different run scenarios for the NCS algorithm. Each preference file had a different representation: p1 represented an IT team's priority to a list of tasks, p2 represent the business teams priority to the tasks, and p3 represented the actual cost of each task. The algorithm has 10 different scenarios which the value *order* determines to run. Each scenario used the preference files, *m* value, and sorting techniques differently. Since we were looking at assigning real work tasks, we wanted to test sorting pieces from greatest to least value so that higher priority pieces would be attended to first. We also wanted to test a starred question from class which asks whether sorting really matters for OCS. The results are posted in the analysis section further down in this paper. The code also contains comments explaining the algorithm and its functions.

The value *PList* captures the last list of variables in the input string which represents the values of platters. The number of items in the list indicates the number of platters the algorithm will create. During the creation, there was some confusion with the team about if the values had to be summed/scaled to equal 1 or if platters can be created and have unassigned pieces left over. Our real world data showed that the total cost for all tasks exceeded the allowed budget for the year. Based on this, that algorithm was programmed to accommodate *PList* in two ways. If the values of *PList* were prescaled and add to be less than or equal to 1, the exact values would be used as inputted in the algorithm. However, if the values summed to greater than 1, then all the *PList* values would be scaled so they would sum to 1. The NCS algorithm basically uses a recursion of OCS where a value popped from *PList* would be used as the *L* value. The left platter created would be the platter for that iteration. The right platter would be passed into the next recursive call along with the next value from *PList*. Regarding the last item in *PList,* it would potentially create two platters if the remaining pieces exceeded its targeted value. In this case, the excess pieces (returned as the right platter) were captured in a *LeftOver* variable. If the pieces were less than or equal to the last *PList* item's value, then 0 was returned in the *LeftOver* variable. The NCS algorithm then returns a list of platters corresponding to the *PList* values and a *LeftOver* list of any pieces that were not assigned. Two versions of the NCS algorithm are in the repository. An initial version is in the *alg 6* fold labeled *NCutsSuffice*V1*.* The final version for this algorithm is in *alg 7* labeled *NCutsSuffice*. The reason for the two versions is that an issue (which will be explained further in the analysis section below) was discovered during testing. Both versions were preserved so that both could be run from the repository to recreate the accounted results.

The second portion of our project began with us building the Moving Knife algorithm. We "move a knife" from left to right over a project composed of tasks in steps of a given resolution. The resolution is assumed to correspond to the smallest allowable division of tasks. In each step, we iterate over the preference lists of the team members still in need of task assignments. If the value of the summed tasks from the previous "cut" to the current location of the knife is greater than 1/n of the whole project value to one of the remaining players, we make the cut at the current location of the knife and allocate that piece to the player who values that piece equal to 1/n within the resolution restrictions. If there is only one player left, we allocate the remaining tasks to that player.

After building Moving Knife, we adapted it to be like the Moving Knife algorithm we learned in class for Near Exact Division, in which envy is limited by epsilon. Our parameters are the list of preference files, the number of people who need to be assigned tasks, the epsilon value, and the resolution. We first divide the project into a list of tasks incremented by the resolution, then we iterate over the list of tasks. For each iteration, we pop the task off the front of the list (this represents moving the knife from the left edge of the task-piece, across it to its right edge) and update our stored values. The stored values represent each player's valuation of all of the

cake currently to the left of the knife. In order for a player to get the cake left of the knife, his or her value of that piece has to be epsilon greater than the value of his/her current piece, and the player must still have a positive number of turns remaining. We store what the “next required value” of the cake left of the knife would be for each player, and compare these values to the players’ values of the cake left of the knife. If a player’s “next required value” is less than that of the piece left of the knife, then that player returns his or her currently owned piece. We put this piece back into the list of tasks and then the player gets the piece left of the knife. The player who just received cake has his or her “next required value” incremented by epsilon, and the number of turns he or she has remaining is decremented by one. We only end the algorithm when all of the tasks in the list have been allocated or when the remaining tasks are less than the “next required value” for all of the players.

Analysis

Our group obtained project and preference information from real world data from our jobs and internships. Michael presented a business proposal from this year that presented 134 requirements that were divided into 7 projects throughout the year. For the overall proposal, Michael presented the business priorities and IT priorities preference files represented below in Images 1 and 2.

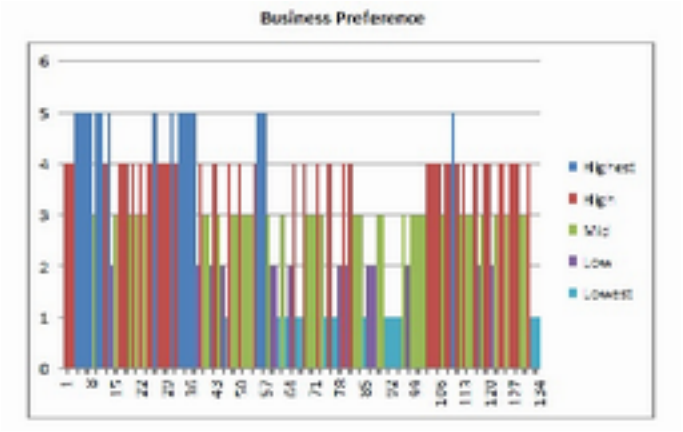


Image 1: Business Preference

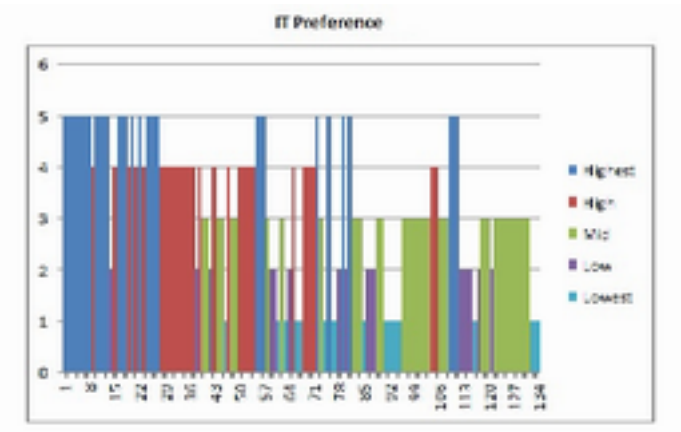


Image 2: IT Preference

All 134 tasks were given a priority based from 1 through 5 with 5 being the highest priority and 1 being the lowest. The IT Preference data is used as the first preference file (p1) for the NCS algorithm. The Business Preference data is used as the second preference file (p2). Additionally, Michael provided a third preference file that represented the cost of each task in units of a thousand (K) dollars, represented in Image 3 below. This is used for the third preference file (p3) for NCS.

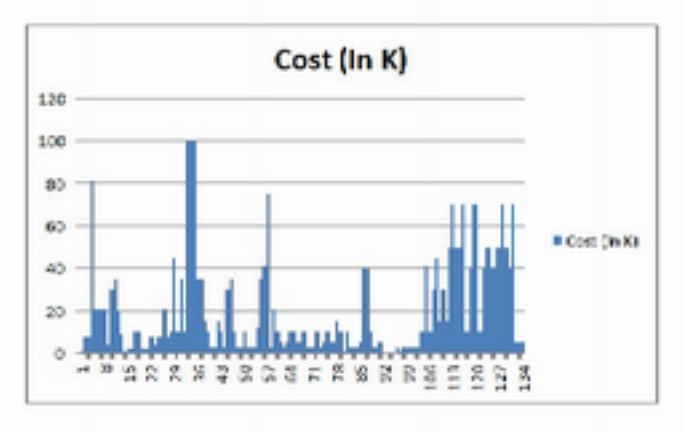


Image 3: Cost of Tasks (Units of a Thousand Dollars)

Additionally, Michael provided the following information listing how the company grouped the tasks (Image 4) and which projects the tasks were completed in throughout the year (Image 5). The task grouping would be used for the basis of the fixed positions in the NCS scenarios 1 through 5 described later in the project analysis section. Note that the value of of the fixed groupings from greatest to least was the same for all the *p1* and *p2* preference files. The actual project assignments would be used to compare with how NCS divided the work across 7 projects for all scenarios.

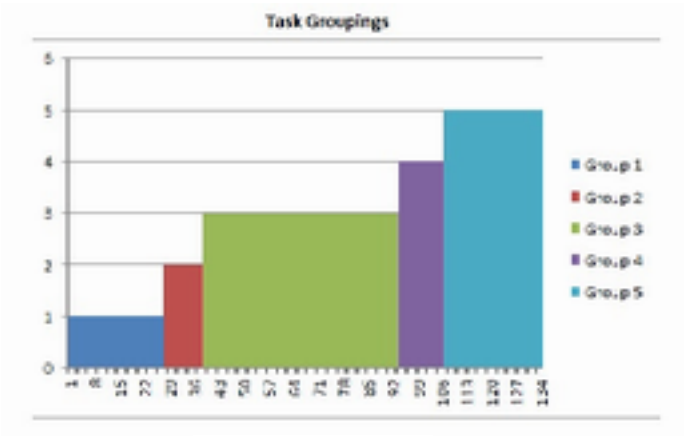


Image 4: Task Groupings

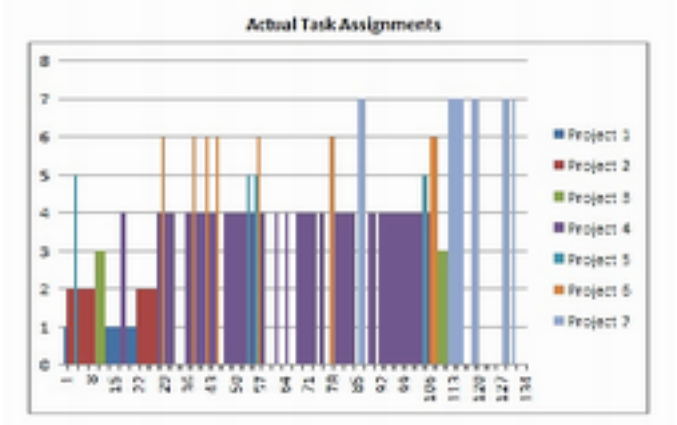


Image 5: Actual Project Assignments

It is important to note that not all tasks were completed in 2012. The cost for all 134 tasks was around $2.75 million. However, the Business team was given a budget of only $1.9 million for the year. Based on these numbers and project actuals, the team generated the values that would be used for NCS *PList* values. The results of NCS would capture which tasks were assigned to which projects (i.e. platters) and which tasks were not assigned (i.e. *LeftOver* pieces*).* The table below (Image 6) shows how the *PList* values were generated for our project.

Project	Budget (Rounded)	Budget / Total Cost	*PList* Value Used
1 – March	$21,000	0.007611	0.008
2 – April	$166,000	0.060167	0.060
3 – June	$165,000	0.059804	0.060
4 – July	$405,000	0.146792	0.147
5 – August	$175,000	0.063429	0.063
6 – October	$287,000	0.104023	0.104
7 - December	$700,000	0.253715	0.254
Carry Over	$840,000	0.304458	0.304
Total (Actual)	$1,919,000	0.695542	0.696
Total (Complete)	$2,759,000	1.000000	1.000

Image 6: Data used to generate *PList* values for project.

With the NCS algorithm, we ran 10 scenarios using various settings for initial pieces, sorting, and player preferences. Note that for all scenarios:

- The same values for *PList* were used for all scenarios as displayed in Image 6 (values .008, .06, .06, .147, .063, .104, and .254). This resulted in 7 platters being created (labeled 0 - 6) along with the *LeftOver* list being displayed.
- All scenarios used a value of 134 for *m* and a value of .0001 for *epsilon.*
- All scenarios used preference files *ncs_p1_it*, *ncs_p2_biz*, and *ncs_p3_it_cost* for preference files *p1, p2,* and *p3* respectively.
- *NCutsSuffices (*in the repository folder *alg 7)* was used for the final results and analysis.

For scenarios 1 through 5, a fixed set of hardcoded initial pieces was used instead of creating *m* pieces based on user input. The fix settings were created based on the Task Groupings describe earlier (see in Images 4 and 6). Image 7 shows how the three preference files prioritized the fixed pieces.

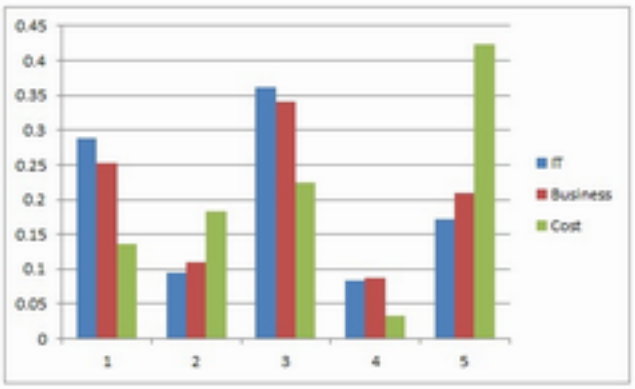


Image 7: IT, Business, and IT Cost Preference On Fixed Pieces

Scenario 6 through 10 would run the same parameters as scenarios 1 through 5 with the exception that the *m* value passed was used to create the initial pieces. The *m* value used was 134 to break up each task into its own individual piece. This simulated how IT and Business groups initially look at the data (in the groups they are presented in and then at the individual level).

For scenarios 1 and 6, the initial pieces were sorted from least to greatest based on *p1*'s preference. After the sort, each piece's value is derived from p3's file to associate a piece to an actual dollar cost. The results of task assignments to projects is shown in Image 8 below.

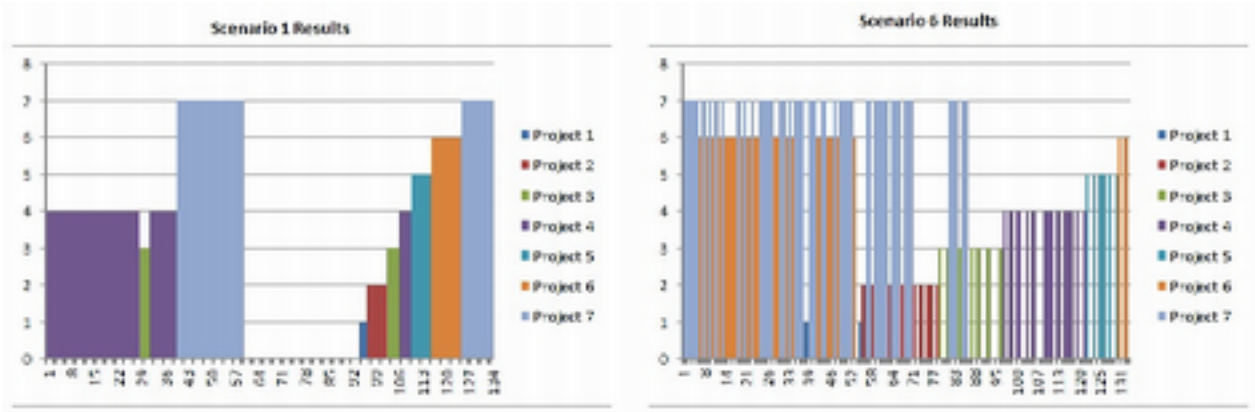


Image 8: Results for Scenarios 1 and 6. P1 Preference with LtG Sorting.

Scenario 1's results were mostly what we expected, since the majority of the Group 3 requirements should have not been assigned, as this was the greatest value piece to *p1.* However based on our calculations, we expected Group 1's tasks to be assigned to a later project. Scenario 1, though, had Group 5 tasks as being a greater value than Group 1. This was verified by a print out of *p1*'s sorted list. We were not able to immediately determine why this happened. This would require us to revisit how we came up with our priority values for any miscalculations and also investigate the effects of scaling the values to 1 or replacing piece values with *p3's.*

Scenario 6's results were mostly what we expected, as the lower valued tasks were assigned to earlier projects and the higher priority tasks were assigned to later projects or not at all. This is verified with tasks 1 through 35 being assigned to later projects and the gaps in tasks 83 through 134 which would represent the high priority tasks in that set.

Overall, Scenario 1 and 6 performed as expected when sorting from least to greatest (LtG). Compared to each other, there is an obvious difference with how assignments are made based on the size and number of initial pieces created. Also both results were very far off when compared to the actual assignments shown in Image 5.

For scenarios 2 and 7, the initial pieces were sorted from least to greatest based on *p2*'s preference. After the sort, each piece's value is derived from p3's file to associate a piece to an actual dollar cost. The results of task assignments to projects is shown in Image 9 below.

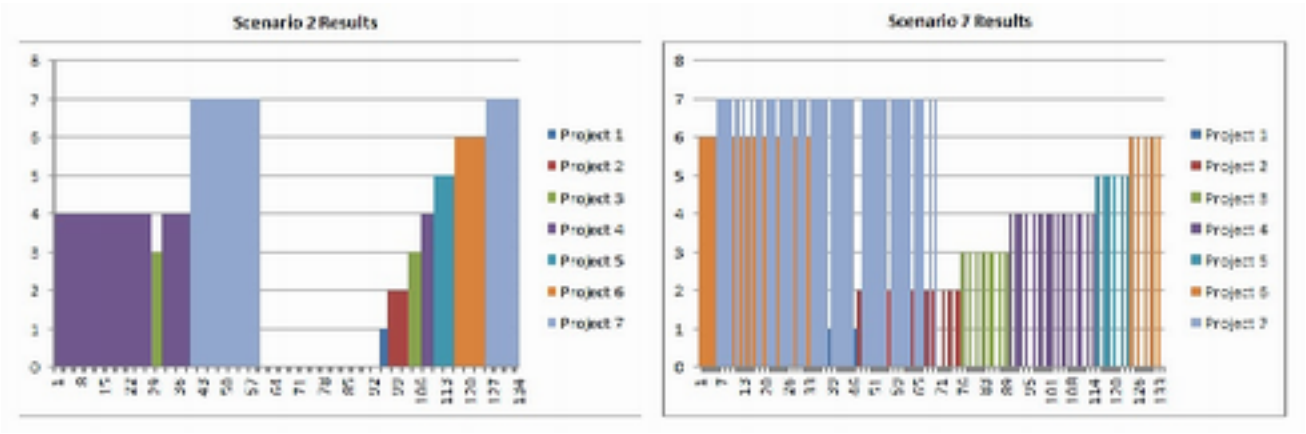


Image 9: Results for Scenarios 2 and 7. P2 Preference with LtG Sorting.

The results for scenarios 2 and 7 were very similar to those of scenarios 1 and 6 respectively. This was expected by our team as the priorities of the *p1* (IT) and *p2* (Business) were similar as seen with Images 1 and 2. Again we noted for scenario 2 that Group 5 had a higher total value with Group 1 which was at odds with our original estimated calculations. Overall, both scenarios 2 and 7 were also far off when compared to the actual task/project assignments from Image 5.

For scenarios 3 and 8, the initial pieces remained unsorted thus having the tasks processed from beginning to end. The values for each piece were pulled directly from the *p3* file. This scenario was mainly used as a sanity test to ensure the initial piece creations did not have any unforeseen effects with the algorithm. The results of these scenarios are shown in Image 10.

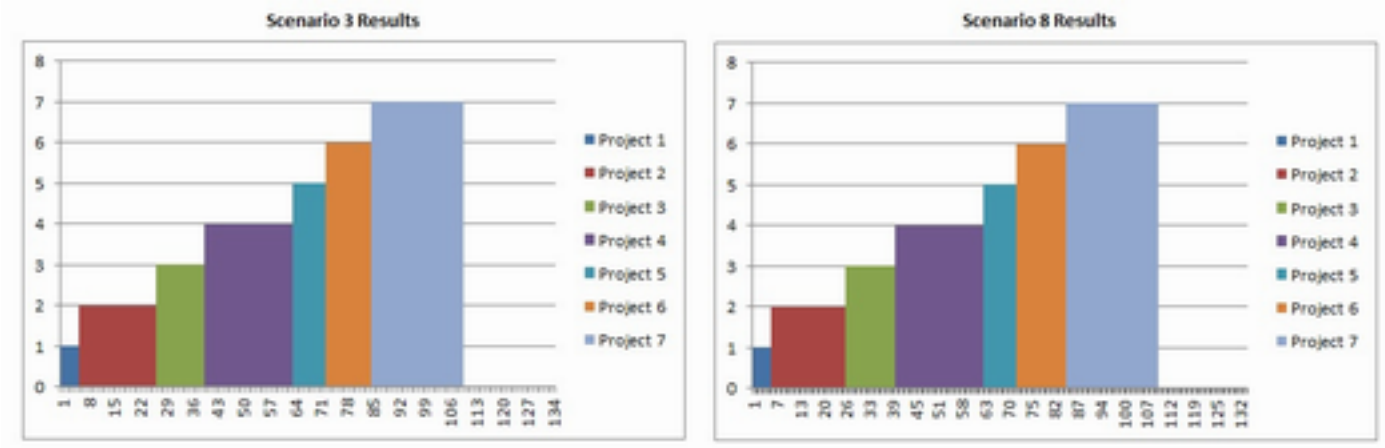


Image 10: Results for Scenarios 3 and 8. No sorting.

The results for scenarios 3 and 8 were what we expected. The assignments should have been linear across the projects with the tail end of the task list remaining unassigned. When compared with each other, the results looked identical which confirmed that there were no issues between the creation the fix and variable initial pieces. The results do not resemble the actual assignments as from Image 5. However, these results were closer to the actual when compared with the previous scenarios tested.

For scenarios 4 and 9, the initial pieces were sorted from greatest to least to (GtL) based on *p1*'s preference. After the sort, each piece's value is derived from p3's file to associate a piece to an actual dollar cost. The results of task assignments to projects is shown in Image 11 below.

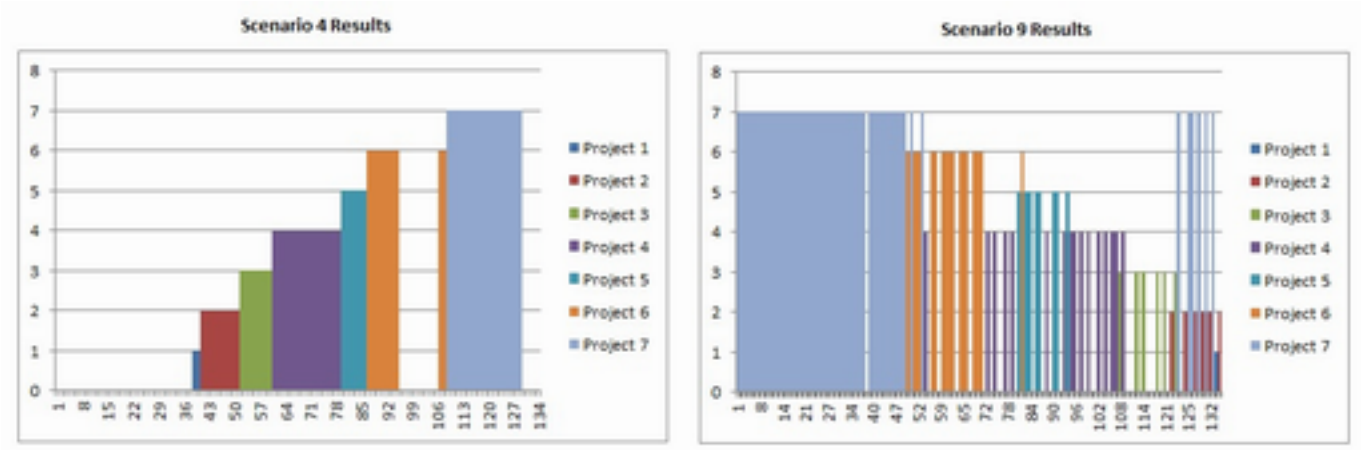


Image 11: Results for Scenarios 4 and 9. P1 Preference with GtL sorting.

For scenario 4, the results were close to what we expected. Again, Group 3 would have been the highest valued piece. Thus the majority of the projects would have tackled that area first which is confirmed on the graph. We expected that the tasks in Group 1 would have been the second priority. However, the algorithm computed Group 5 as again having a greater value than Group 1. Given that result, Project 7 was assigned most of the later tasks which were the more expensive ones as seen in Image 3. Thus, we were not too surprised that Group 1 was not assigned, as the last set of tasks would quickly add up to the projects' budget limits.

For scenario 9, we viewed this as an unexpected result. Image 2 shows that IT's highest priority tasks ranged between tasks 1 through 60. So we expected that the first set of projects would be assigned tasks from the first half of the list. The results appear flipped as the earlier projects tackled more of the later tasks. We believed that this could be related to a similar issue with how the Group 5 tasks had higher value than the Group 1. We would need to investigate this item further as we did not see any stand out issues with the code.

Overall, scenario 4 is vastly different from both scenario 9 and the actual project assignments. The results from scenario 9 also do not match with the actual project assignment. Our expectation though was that scenario 9 would have a result that would have been much closer to the actual task assignments.

For scenarios 5 and 10, the initial pieces were sorted from greatest to least based on *p2*'s preference. After the sort, each piece's value is derived from p3's file to associate a piece to an actual dollar cost. The results of task assignments to projects is shown in Image 12 below.

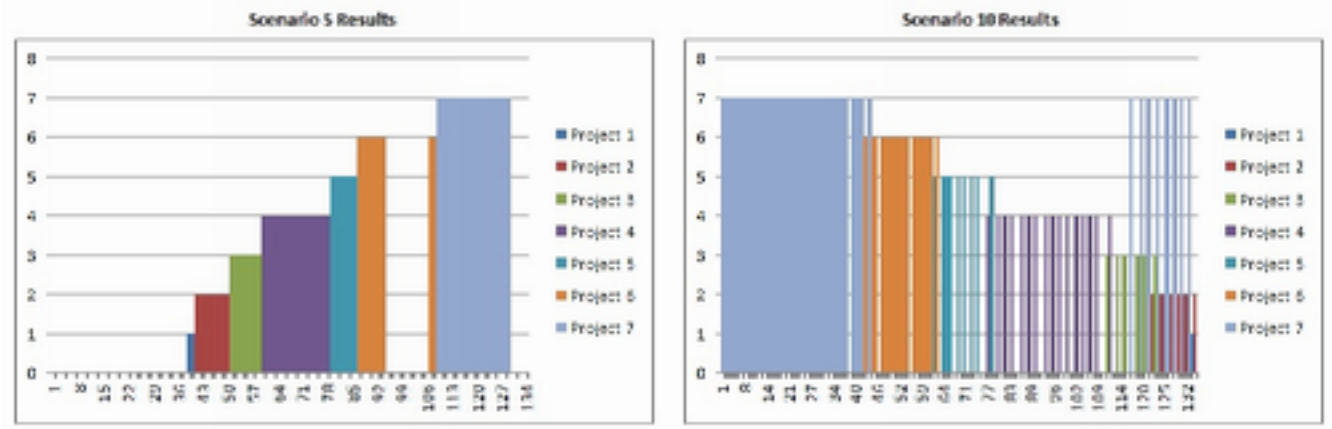


Image 12: Results for Scenarios 5 and 10. P2 Preference with GtL sorting.

The results for scenarios 5 and 10 were very similar to those of scenarios 4 and 9 respectively. This was expected by our team as the priorities of the *p1* (IT) and *p2* (Business) were similar as seen with Images 1 and 2. Again we noted for scenario 5 that Group 5 had a higher total value with Group 1 which was at odds with our original estimated calculations. We also again noted that scenario 10 had similar unexpected results like scenario 9. We did expect to see earlier projects handle more of the last sets of tasks as the business did have more higher priority items in that area (see Image 1). But we were expecting earlier projects to handle the majority of the tasks from 1 to 60 as that is where the business had its second highest group of priority items. Overall, both scenarios 5 and 10 were also far off when compared to the actual task/project assignments from Image 5. We expected that scenario 10 would have been closer to the actual results.

We discovered one unexpected item with an earlier version of our NCS algorithm. We initially ran scenarios 1 and 3 against *NCutsSuffice V1* in *alg 6*. The results from this test are shown in Image 13.

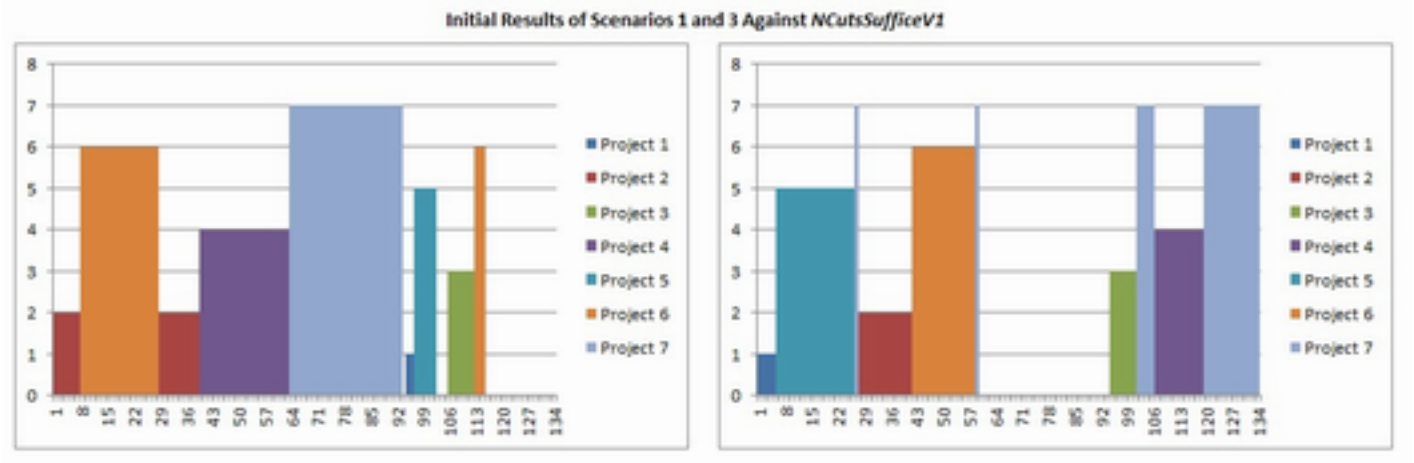


Image 13: Results for Scenarios 1 and 3 from Testing on *NCutsSufficeV1*.

The initial results were unexpected, especially with scenario 3 as we thought we would have seen a linear assignment of the tasks like in Image 10. Upon further investigation, we determined that the results were due to how we originally assigned piece *b* from the *PieceToCut* on the right platter. In *NCutsSufficeV1*, we stored piece *b* at the end of the right platter list instead of placing it as the first item. In OCS, this would not matter as we only returned two platters. But in NCS, the right platter was recursively being passed back into the algorithm to create additional platters. Since *b* was being placed at the end, the newly cut pieces would not be accounted for until the end of the algorithm. This caused the results seen in Image 13. We corrected this by appending any *b* pieces to the front of the right platter. This way, the cut pieces will be handled first before moving on to the next initial piece. Our finding leaves room for further research on how sorting cut pieces could affect platter creations. We believe this could be applicable in the real world setting where parts of cut tasks may be reprioritized into the list with a lower value. We left *NCutsSufficeV1* available in the folder *alg 6* to recreate these results and for possible future comparison with *NCutsSuffice.*

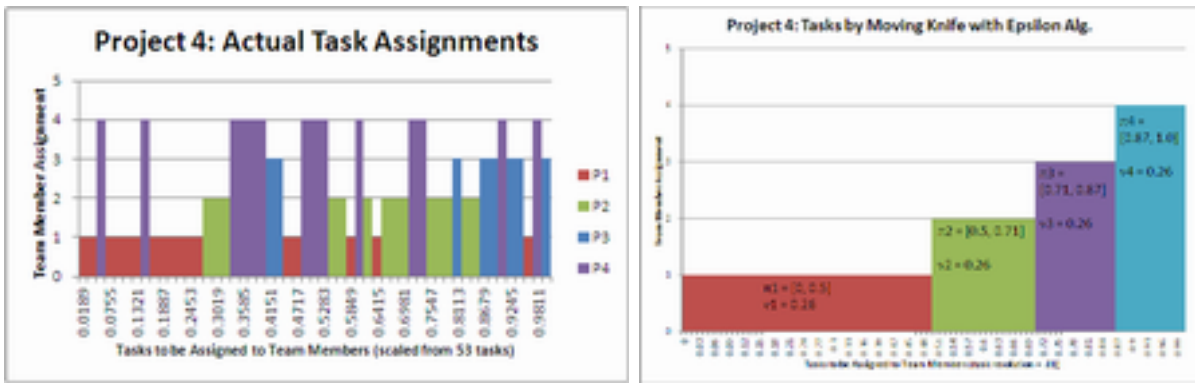
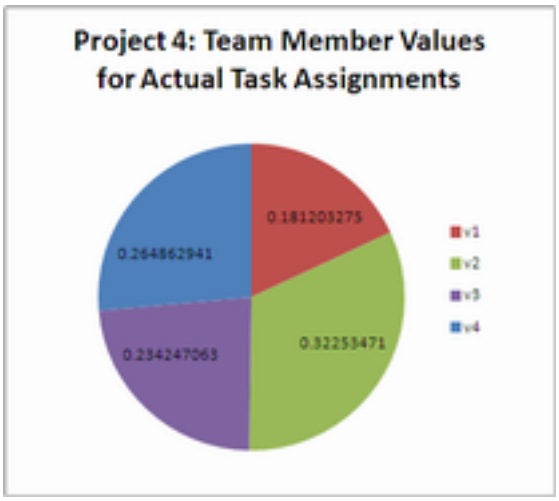


Image 14: Actual division of tasks in Project 4

Image 15: Division of tasks in Project 4 using MKE

The preference files dev1 through dev4 are representative of the preferences of four team members who were assigned to project 4 at Michael's company. Each file lists the 53 tasks that were included in Project 4 and one person's ranking of how much he or she desired to work on each task (see Image 14). Tasks were rated with values 1 through 3 with 1 being "uninterested" and 3 being "I'd love to work on this". The distribution of tasks shown above in Image 15 is the result of running MKE on the preference files.



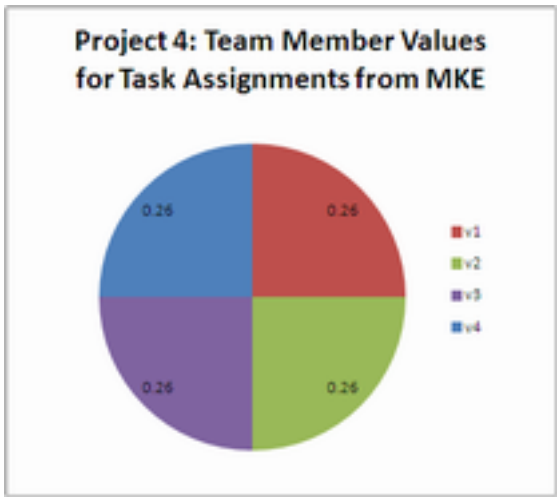


Image 16: Valuation of actual assigned tasks

Image 17: Valuation of assigned tasks using MKE

We took the actual task assignments from the company's project and created lists of the tasks assigned to each of the four team members. We then ran an analysis script on the lists that used the dev1 through dev4 preference files to sum each team member's values for the tasks that were actually assigned to them. This is shown above in Image 16 along with the team members' valuations of their assigned tasks as assigned by our MKE algorithm in Image 17. The assignments that resulted from our MKE algorithm give team members a group of tasks that they each value more highly than 1/n of the tasks, which means that they each received more tasks that they were excited about.

We built Moving Knife (MK, *alg 4* in our repo) to be used to divide up tasks between members of a development team after determining the group of tasks assigned to that project using the NCS algorithm. We modified it to be the Moving Knife with Epsilon algorithm (MKE, *alg 5* in our rep), such that all players' envy is limited to at most epsilon. Once we completed the MKE algorithm, we found that for some sets of preference files (though not for the dev files representing the actual Project 4), some pieces of cake were not being allocated to any players. We were concerned, because even though every player ends with at least 1/n of the cake (preserving proportionality), the tasks (pieces of cake) in our scenario all needed to be completed for the project. Thus all needed to be assigned to a team member.

After some tinkering, we realized why there was sometimes unallocated cake at the end of the algorithm's run. In the MK algorithm, the last player to receive cake just gets all the cake remaining to the right of the knife when the second to last player says "stop". This is because the last player never said "stop". In the MKE algorithm, the scenario often occurs where all players have said "stop" and received cake but are still playing because they could improve their share. The cake to the left of the knife (knife moves left to right across cake) has to be epsilon greater in value than the piece a player currently has. So if there is cake remaining but it is of less value to all players than [epsilon + the value of their current piece], it will remain unallocated. No player will trade in his or her piece for the remaining cake. Another possible reason for the unallocated cake may be due to how Python rounds numbers.

Although algorithmically it is okay to have unallocated tasks on the project, it is not ideal in business scenarios to leave parts of a project unallocated. We assume that in a real world situation, these tasks would be dealt with by either assigning them to the group members already assigned to the most similar tasks, random

assignment, or incorporating another fair division algorithm. Maybe the team leader or supervisor would assign the tasks, or perhaps team members could work collaboratively on them. We decided that assigning the small number of unallocated tasks would be outside the scope of our project.

Conclusion

In conclusion, we believe our project should mix results for meeting our intended goals while opening new areas for further research and development. The NCS algorithm and the scenarios tested did not replicate the actual task assignments and had unexpected issues with sorting priorities. However, our projected showed that the size of initial pieces and sorting orders could greatly impact final results. Further research can be done to determine how sorting orders, initializations, and prioritizations can be further enhanced to accommodate complex scenarios in the real world. The MKE algorithm did meet our goal by showing it could assign work more equally and fairly than the real world results. However, finding that our algorithm did not always assign all the cake led us to try to explore whether or not full assignment of the cake is an actual requirement for envy-freeness in all situations. Certainly these results show that the algorithm is not optimal, or could be combined with another algorithm to deal with the unassigned pieces. Overall, we met our intended goals and learned a lot about what can be done with fair division algorithms in real world settings.