AI Scheduler

Ophir Carmel
Hebrew University of Jerusalem
ophir.carmel@mail.huji.ac.il

Matan Epel
Hebrew University of Jerusalem
matan.epel@mail.huji.ac.il

Amit Roth
Hebrew University of Jerusalem
amit.roth@mail.huji.ac.il

***Abstract*—In our project we will solve the multi participants schedule problem - generate calendars for several users with mutual events. We used variety of topics which we learned in the course, such as CSP (Constraint Satisfaction Problem), game theory, genetic algorithms, hill climbing, and neural networks. Our final system wraps our algorithms with a graphical user interface and communication with google calendar.**

I. Introduction

Everybody has basic tasks that they need to do, side projects that they want to work on, meeting they must go to, and so on. Scheduling your meetings with others is a time consuming task, and often most of our meetings are delayed or canceled, and for some users the time is not possible or just not so convenient. Our project comes to solve this problem, and to supply a mutual and fair calendar for all users.

II. The Problem

As a complete system - our project solves several related problems.

A. Define the scoring system

This part seems trivial but it isn't. The problem of defining the scoring system includes taking the users preferences and giving a score to a given schedule. The user preferences as we defined them was based on 5 criteria:

1) close meeting - meetings are one after the other
2) close tasks - tasks are one after the other
3) long breaks - making the breaks between events as long as possible
4) starting late - starting the day late
5) finishing early - end the day early

We wanted to give the user the option to give weight for each of the criteria and by that he create his preferences. In the solution part we will describe how we calculated the score in order for the next chapter of optimizing on the score to be possible.

B. Find an optimal assignment of user's tasks

Given n users $\{U_1, \ldots, U_n\}$ and a corresponding constraint $\{C_1, \ldots C_n\}$ for each user. Each of the users has tasks which need to be scheduled, $\forall i \in [1, n], \{T_{i,1}, \ldots, T_{i,t_i}\}$, which t_i stands for the number of tasks of the i-th user. We also given a set of scheduled meetings $\{M_1, \ldots, M_k\}$ and for each meeting a set of participants $\forall i \in [1, k], \{U_{a_{i,1}}, \ldots, U_{a_{i,m_i}}\}$ which m_i stands for the number of participants of the i-th meeting. Finally, we are given a set of scheduled "must be" (time slots for each users were he must be attended in) $\{MB_1, \ldots, MB_n\}$. We need to find a satisfiable assignment to the times of the user's tasks, taking into consideration each user's constraints and preferences and the meeting and must be times were he can't do tasks in (unless he define otherwise in his preferences). Then, after we found an assignment, we need to find an optimal one - a satisfiable assignment that maximizes the preferences of the user.

C. Find an optimal assignment of the meetings

Given all the meetings of a group of users - we need to find them a time assignment which is optimal for all the users. Most of the times, plenty of satisfiable Assignments exist and we want to find an optimal assignment, which all of the users are happy with, based on their preferences. In order to talk about the "optimal assignment" we need to define it. As proposed by Nash [2], we can use the Nash equilibrium to define our optimal assignment. In our case, the Nash equilibrium is a time assignment for the meetings where the optimal solution for the tasks of each user can't be better if we change it's meetings times (and to simplify things - change one of it's meetings time). Obviously, finding a Nash equilibrium is extremely hard (NP hard to be specific, or even impossible when there is no equilibrium), and we would like to approximate the equilibrium using "less strict" evaluation functions for the problem that will calculate a score based on how approximately close we are to the equilibrium (or other "game theory" spots. For example - maximizing on the sum of the group). In the solutions section we will discuss those functions.

D. Classify Event

In our project, we defined 4 kinds of events:

1) Tasks - Basic events, everyday stuff.
   Some examples for the title of those events include: 'Do homework', 'Play sports', 'Learn about {random topic}'...
2) Meetings - An event which must shared by multiple participants.
   Some examples for the title of those events include: 'Meeting with Amit', 'Amit and Matan', 'Meeting about {random topic}', 'Do homework with {random name}'...
3) Must Be In - An event that requires attendance. The time slot for this event cannot be changed, and other meetings and tasks cannot be scheduled at the same

time as this schedule. Basically, a block in the calendar that cannot be overridden. Can optionally be shared by multiple participants. Some examples for the title of those events include: 'Lecture about Topic', 'Boss', 'URGENT - discussion about {random name}', 'Exam', 'Nap', 'Exercise'...

4) Lunch - self-explainable. Lunch break.

Instead of the user specifying for each event what type of event is it, we decided to train a classifier that will do it automatically. The classifier will get the event name, and return its type - "TASK", "MEETING", or "MUST BE IN" (we do not include lunch in our classification problem) in order for the program to know what data to ask for (other participants when it's a meeting, start time when it's a must be and so on). In the solution section, we will discuss our different approaches to this problem.

III. Our Solutions

In our project we used a variety of AI algorithms learned in class and over the internet. The algorithms are: optimized CSP, hill climbing, genetic algorithms, and neural networks.

A. Define the scoring system

The scoring system is based on the 5 different parameter defined above. The approach we took in order to score those parameters is as follows:

1) close meeting - for every 2 continuous meeting, we credit 1 point
2) close tasks - for every 2 continuous tasks, we credit 1 point
3) long breaks - we gave 1 point for every hour of the longest continuous break in the day
4) starting late - we gave 1 point for every hour of delay after the start of the working day
5) finishing early - we gave 1 point for every hour of early termination of the day before the end of the working day

However, these scores gave much more weight to the last 3 parameters due to the typical combination of meetings and tasks. This problem makes the weight the user choose useless. So, in order to solve this - we normalized each parameter score by the maximum possible score.

B. Scheduling Each User's Tasks

This section is part of the total problem of finding the optimal assignment of user's tasks.
In order to schedule each user's tasks, we solved the problem as a CSP (Constraint Satisfaction Problem). for each user we have a set of variables $V = \{T_{i,1}, \ldots, T_{i,t_i}\}$, a domain of times $D = \{< h, m > | h \in [0, 23], m \in \{0, 15, 30, 45\}\} \times [1, 7]$ which stands for the time and day in a week, and a function of constraints C which contains the "hard constraints" that defines if certain assignment is valid or not. We solves this problem using backtrack and a simple heuristic that chooses in every iteration, the variable (task) with the largest duration and assigns time for it. The longest events are the hardest to schedule hence, dealing with those first made out backtrack algorithm super quick.

C. Find an optimal assignment of user's tasks

When taking the soft constraints into consideration, our problem shifts from a CSP problem into a CSOP (Constraint Satisfaction Optimized Problem). CSOP is a generalization of CSP - it has an additional function f which evaluates the score for every final solution. In our case, every schedule has a score based on the user's soft constraints. The mathematical definition of CSOP is identical to CSP, with the addition of the function f that returns a score on any valid assignment.
Solving a CSOP is a hard task because not only do we need to find a solution, it has to be an optimal one as well - find the single assignment which maximizes f. We can solve it utilizing heurestics and pruning techniques, as described in the paper 'Why is Scheduling Difficult? A CSP Perspective [1]'. However, we won't get a satisfactory running time because this task will get executed many times by the "Optimal Assignment of Meetings" algorithm. We wanted to supply a solution in a reasonable amount of time, so instead of solving the CSOP problem we approximated the solution by solving the CSP a multiple amount of times, starting from different randomized states. Then, we choose the best solution out of all the runs, which approximates the CSOP with enough runs. In addition, in some cases the CSP takes too long, so each call is executed with a timer that halts the function when it times out.
The algorithm is as follows.

1) Shuffle the domain
2) Backtrack, prune the call of the function if exceeds a predefined time
3) iterate several times, save the returned values and chooses the maximum

As a result of the shuffle of the time domain each run, we are able to achieve good results, and taking a good result with a slight number of different searches. After this process we are left with a satisfiable assignment for all of the users, which also semi-optimizes it soft constraints. this procedure returns the assignment and the score, calculated by f, which will take a big part in solving the "Optimal Assignment of meetings" problem.

D. Find an Optimal Assignment of meetings

This part assumes we can get an optimal assignment for the tasks of each user given the meeting times and must be times (as described in the previous chapter). Now, we need to find the optimal times for the meetings in order to find the optimal total solution. The term "optimal" is a bit complicated in this case, and in this part - we assume we have a function that given the scores of all the users, returns a total scores that making it bigger will make the solution more optimal (we will discuss how we define the formula for this score in the next chapter).
The route we took for optimizing on the optimal solution was based on two main (classes of) algorithms:

1) hill climbing (a variant of hill climbing)
2) genetic algorithm

In the following paragraphs we will describe the solution in each of these two algorithms.

Genetic algorithm: The basics of genetic algorithms were learned in the course. The main parts of an algorithms are as follows:

1) starting from X different starting points (aka assignments of time for the meetings)
2) calculating the scores of each one of the samples.
3) populating the K best samples (based on score) using a combination of making "children" out of pairs of samples and then giving each one of them a "mutation".
4) returning to step 2 recursively until we have reached certain amount of steps.

So all we had to do is to define the "children" function and the "mutation" function:

1) children - we took each pair of samples (aka - 2 vectors of time assignments for the meetings) and for every meeting we replaced the times between the vectors. For example - $[t_1, t_2]$ and $[t'_1, t'_2]$ will have the following children: $[t_1, t'_2], [t'_1, t_2]$.
2) mutation - the mutaion was pretty easy, we took one random meeting and changed her time to random time (obviously - time out of the times satisfies the constraints of all the participants)

To define the algorithm completely, we just need to define how we pick the starting point. After trying some heuristics we concluded that picking the starting points randomly will gives us the best results.

Hill climbing: As we all know, hill climbing works on continuous parameters and our problem is obviously discrete (discrete number of meetings and discrete amount of time slots - we had slots of 15 minutes each). So, we didn't actually used a hill climbing, but a form of greedy search which tries to maximize the score of the states. The steps of the algorithm are as follows:

1) start from a random starting point
2) calculate the score of all the neighbors
3) go back to step 1 until the score is not getting better

The main work in this algorithm was to define the neighbors of a state (a time assignment for the meeting). We took 2 approaches and we'll describe them both:

1) permutation - the basic method was to take the vector of time slots and to switch between the times of 2 meetings. The neighbors of the vector will be all the options for those switches. For example, the neighbors of $[t_1, t_2, t_3]$ will be $[t_1, t_3, t_2], [t_3, t_2, t_1], [t_2, t_1, t_3]$.
2) all options - the more complex method (which resulted in much more neighbors) was to iterate over the meeting and each time to replace the time of 1 of the meeting to all of its possible times. An important note is that for each neighbor we replace only one of the meetings times, but we create the neighbors for all the meetings.

These neighbor methods are supposed to give us a little change from the current state and hopefully one of the neighbors will give us a better solution.

The last part we need to discuss about in the algorithm was epochs. As we all know, hill climbing tends to converge into local maxima which isn't really good for us. The known solution is to give a chance of ϵ to go to a neighbor other then the best one. But, in our specific problem the amount of steps in order to get to the maximum is low (linear to the amount of meetings) but a step time is high. Those restrictions results in making the ϵ solution irrelevant. So, our solution was trying X number of epochs, when each time we start from another starting location and taking the maximum of all the epochs. A quick note - in the first neighbor method - we need more epochs since the meeting time doesn't (only the duration in each time).

E. What is optimal?

As we promised in the previous chapter - we need to explain our scoring function which results with one of the following 2 game theory wishes:

1) maximizing on the group score
2) finding the equilibrium point

The scoring function we wrote takes the scores of each user and returns a weighted score.

1) group score: Maximizing on the group score is pretty straight forward - we just take the average/sum of the scores and return it.

2) equilibrium score: In this part, we tried to create a scoring function that as bigger as she gets - we get closer to the equilibrium point. However, we have two problems. The first one is that probably there isn't such a point. The second one, even if there is - it's really hard to find it, and defining such a function is nearly impossible.

The approach we took in order to solve these problem was as follow: we can't know if there is an equilibrium point, surly not how close we are to it, but we can define a new parameter of how "fair" the current point is. We defined "fair" by saying that most of the users are close to the average score. So, the initial thought was to take the standard deviation of the scores and trying to minimize it. However, this will create "fair" solutions, but not good on average (we are not maximizing on the score). The solution was to take into consideration the average of the scores too and dividing it by the std (in order to credit high scores, but to "punish" them if they are not fair). We ended up with the following formula:

$$\frac{average(scores)}{max(0.1, std(scores))}$$

The 0.25 is placed there in order to take care of the formula highly preferring scores with low std.

F. Classification of Assignments

To approach the classification problem, we first needed data. After some online research, we were unable to find datasets that include many event names from google calendar, and so we decided to generate the data ourselves.

1) Data Generation: If we want to generate data, we must use a lot of randomness. To craft unique names for events, we use a random name generator in python, and a random topic generator. The names were generated using a package we found online, and the topics were generated by choosing at random from a list of topics. Now, to generate the data, we built templates for each one of the classes. For example, a template for a MEETING would be 'Meet with {random name} and {random name} about {random topic}'. We made as many templates as we could think of for each one of the classes, so the classifier trained over the generated data would be able to generalize over real data as well. In addition to the data generation, we created a smaller handcrafted test set which wasn't created using templates, to see if the classifier was able to generalize.

After generating the data, we need to train a model. We trained three different models. Disclaimer - this part has concepts that the course didn't cover, but we wanted to put the extra work into understanding them and implementing them because they were interesting and cool.

The models we used were:

1) Logistic Regression - a model that utilizes logarithmic functions to classify the names. This solution did manage to get high accuracy on the training and evaluation, but when evaluating over the handcrafted test set, it doesn't get high accuracy. This is probably caused because the model takes the whole string as an input, and doesn't break it to tokens, and therefore cannot generalize very well on inputs that are not generated from a template.
2) Neural Network - a simple model containing a single fully connected hidden layer with RelU activation, a dropout layer, and an output layer with the softmax activation function and 3 neurons - one for each class. Before entering the network, the string input gets tokenized, and the input is turned into vector of integers, by the words in the string. As a result of this, the model is able to generalize much better - even one word from a template is enough to do determine the class. When evaluating the NN on the handcrafted test set, the results were very good - it was able to generalize well on inputs that are not generated from the templates.
3) Convolutional Neural Network - a neural network with a convolutional layer, and an output layer. Similarly to the previous model, before entering the network, the string input gets tokenized, and the input turns into vector of integers, by the words in the string.

Each of the models reach a high degree of accuracy training over the dataset and evaluating over the generated test set, but they reach varying degrees of accuracy over the handcrafted test set, which we will discuss in the next section.

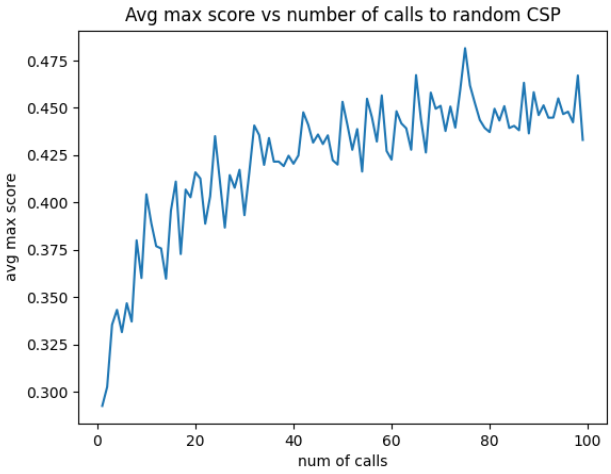
IV. Benchmarks

Firstly, we need to describe how we've measured the benchmarks of each part. We will describe the criteria for the following problems:

1) optimal assignment of user - we will show a graph of the average max-score we get from our assignment versus the amount of calls to the random CSP solution. We will plot 5 must be events and 10 tasks. The user soft preferences are starting the day late and nearby tasks.
2) optimal assignment of meeting - in order to test our meeting scheduler we created a basic formation: two users, 5 joint meetings and 5 tasks each (changed in the running time benchmark). We created a difference in their preferences - the first wants the meetings to be close together, the second wants to start the day late. We will present the following benchmarks for each of the algorithms (hill climbing, genetic algorithm) - score by amount of epochs (genetic) or steps from start (hill climbing) and running time by amount of meetings (plus the time by amount of starting points for the genetic algorithm). In addition, we will show the difference between the "sum" and "equilibrium" scoring functions on a more complex formation.
3) classification - we will look at the training and test groups correct classification rate. Plus, we will create a test data that was not generated by computer (but by hand) and check the correct classification rate over it.

A. Optimal assignment of user's tasks

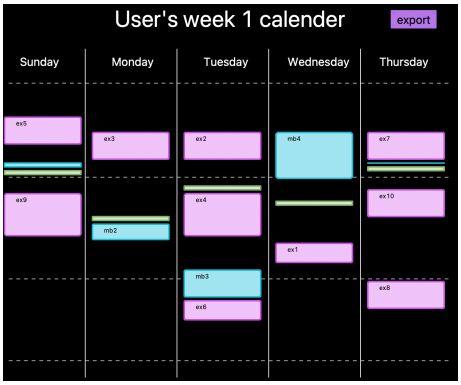
In the following graph we can see that the average score of X runs converges as X getting bigger.



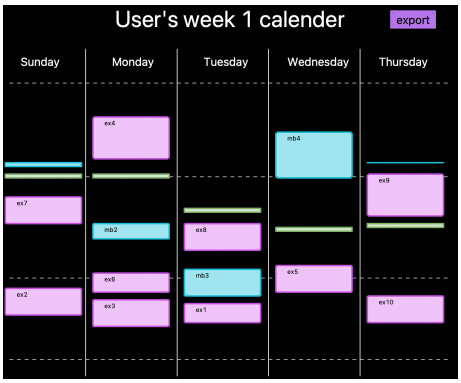
The results are as expected since we are taking the maximum over an ascending number of runs, and each run will contribute a lowering chance of improving the score as the number of runs ascend.

We wanted to "measure" the score visually, so here are the schedules for 1 run and for 100 runs (for the user preferring to start the day late). The results are:

After one run:



After 100 runs:

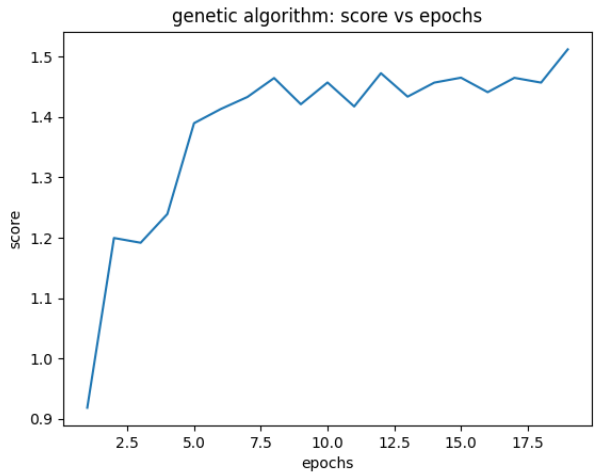


We can see the obvious difference - one run gave us pretty random schedule, as opposed to the 100 runs which tried to start the day as late as possible (the blue events are a must be so their location could not be changed).

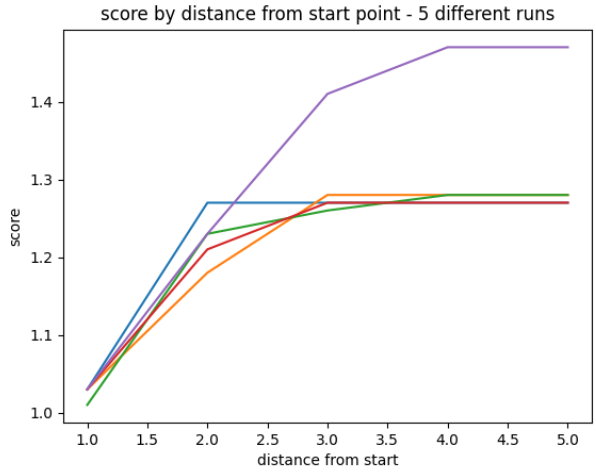
Another interesting observation is about the use of the meeting algorithms on top of our approximation. If we take one user only (with meetings) and let the scheduler schedule - the results are much better. For example (also prefering starting late but adding meetings):

B. Optimal assignment of meetings

1) Scoring results: In the genetic algorithm we've got the following weighted scores as a function of the amount of epochs:



When running with the hill climbing algorithm, we've got the following weighted scores as a function of the amount of steps from the beginning:

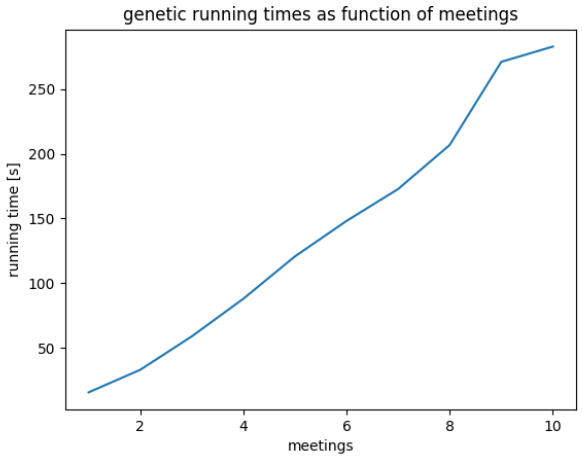


The genetic algorithm's results are clear - as we use more epochs - the score ascend, and converges to a maximum when the amount is getting too big. The hill climbing results are a bit more interesting. We can see it converges to a maximum at about 3-4 steps, which makes sense since we have 5 meetings to schedule. But, we can see that most of the time it converged to a local maxima!

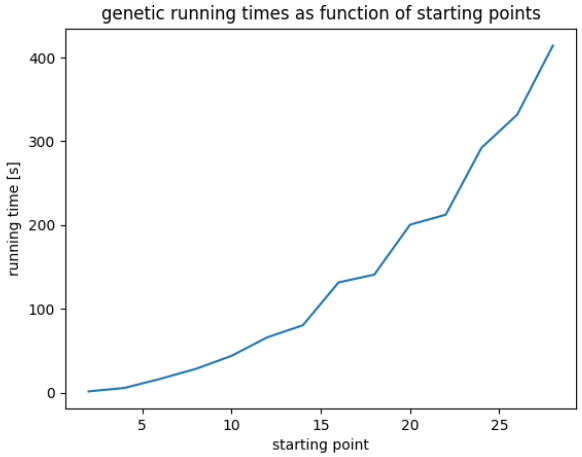
When comparing the algorithms we can see that the genetic algorithm gives us better results, unless the hill climbing algorithm does not get stuck in the local maxima. The conclusion for the final program was to run the hill climbing algorithm for more than 1 epoch in order to escape from the local maxima.

2) Running times: We wanted to measure the amount of time it takes to schedule using the genetic algorithm, and the hill climbing algorithm, For the genetic algorithm we looked at times as a function of starting points, and in the hill climbing we looked at times as function of amount of meetings needed to be scheduled.

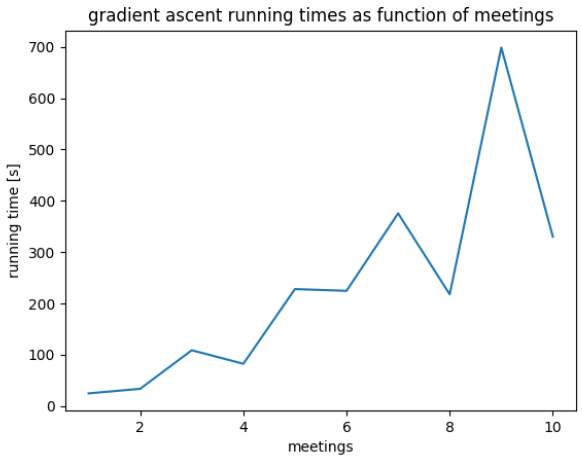
The results for the genetic algorithm are (function of meetings):



And for the amount of starting points:



The results for the hill climbing algorithm are:

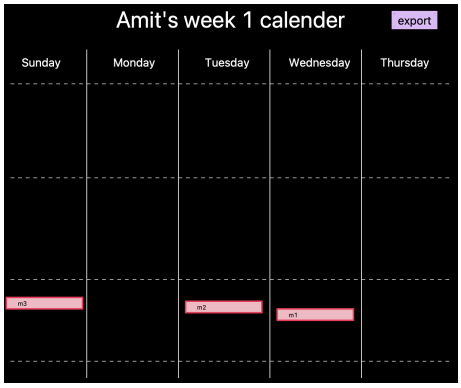


As we can see, the running time of the hill climbing is square in the amount of meeting. The anomalies in the graphs are from runs where the amount of steps needed to reach the maximum point was low (a lucky starting point).

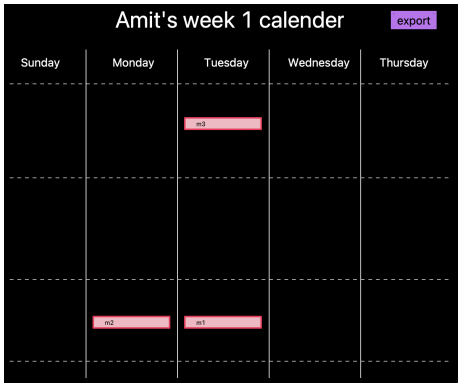
The running time for the genetic algorithm was linear in the amount of meetings (due to the fact that the amount of children created is linear in the amount of meetings), but square in the amount of starting points (since the amount of children is square in the amount of parents).

Comparing between the algorithms analytically in measures of running times is difficult due to the different structure of assignments we used for every test (it was done in order to keep us at reasonable running times). But, if we look at the amount of epochs it takes for the hill climbing to converges to the maximum - the genetic algorithm running times are much better.

3) Sum vs Equilibrium: In order to test our sum and equilibrium functions and compare between them - we wanted to create a test set which will make it easy to see the differences (note that the code works as well on more complicated test sets but it's much harder to see an obvious difference between the two functions). The test set we decided to use was based on meetings only - 3 shared meetings of an hour between two users (without any tasks and lunches). One of them prefers to start the day late, and the other wants to have long breaks, and we wanted to test the performances of our meetings scheduler. As we saw - the genetic algorithm is the one we want to use, and we let it run with a lot of epochs and starting points in order to find an optimal solution.
The results we've got when trying to maximize on group score are:



The results for trying to maximize on equality are:



As we can see, when maximizing on sum the algorithm chose to maximize on the first user only (start late) and didn't care about making long breaks (because when maximizing

on sum it doesn't really matter). But, when maximizing on equality - the algorithm moved two meetings on one day together in order to create long break on Tuesday.

C. Classification

Ophir - add benchmarks

V. Summary

In our project we wanted to create a complete calendar scheduling system. We started with approximating a solution for CSOP problem using backtracking in order to create a schedule for one user. Then solved the shared meetings scheduling problem using two game theory approaches - maximizing for the group and for equality, which we have created a suitable maximization function for. And then used the function using a version of hill climbing and genetic algorithm in order to find the solution. Finally, to complete our system, we used machine learning algorithms (neural nets and logistic regression) in order to predict the type of events in our schedule.

References

[1] Mark S Fox and Norman M Sadeh. Why is scheduling difficult? a csp perspective. In *ECAI*, pages 754–767, 1990.

[2] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

VI. Appendix A - running the code

In order to run the code - simply follow the steps in the README.md file.

VII. Appendix B - Search Domain Approximation

We splitted our time in a day into slots of 15 minutes, hence we get $d = 24 * 4 * 7 = 672$ slots in a week. each meeting can be placed in each of the time slots, hence for k different meetings we will get d^k options. Now, assigning t tasks for n users gives us the maximum size of our search problem - $d^{k \cdot n \cdot t}$.

VIII. Appendix C - The Project Code

All of the code is hosted and publicly shared on Github. you are welcome to use it. An important note is the GUI was written on Macos, so it may be less pretty when you run it on Windows or Linux.

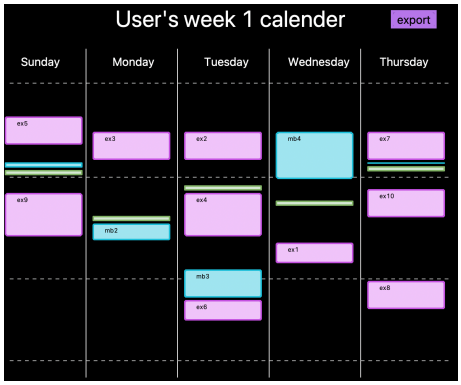
IX. Appendix D - Features

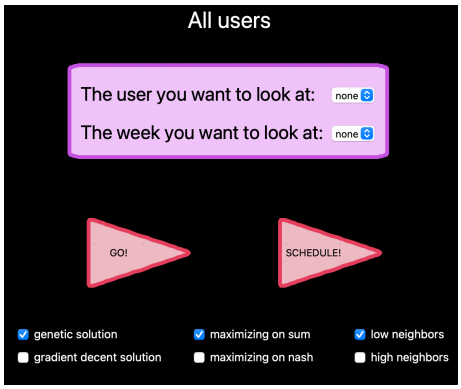
A. Graphical User Interface

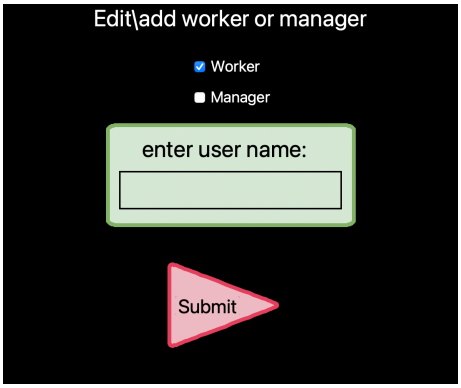
Since we wanted to create a complete environment for scheduling - we had to create GUI for the users to use. The GUI includes the following screens:

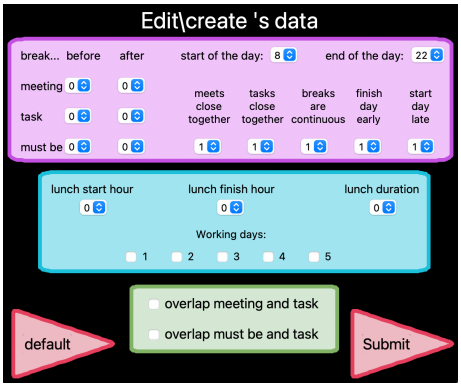
1) home page
2) adding user (and his preferences)
3) adding tasks/meetings/must be for a user
4) scheduling page
5) calendar page for each user

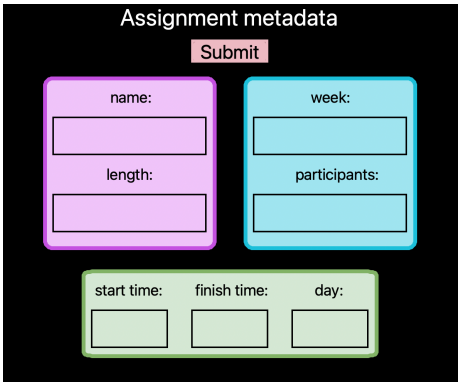
Here are screenshots of the GUI we created:

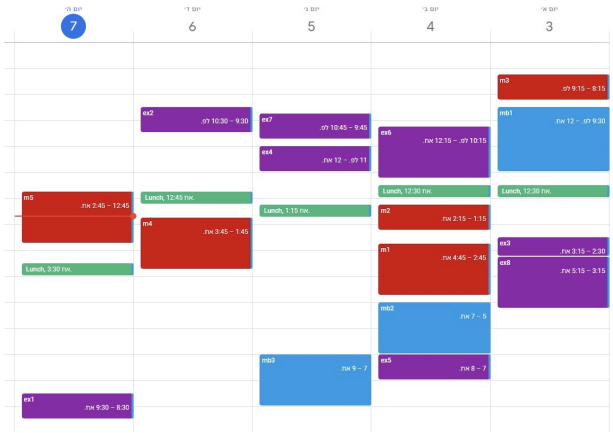












B. Google Calendar API

We added a feature that enables compatibility between our project and Google Calendar. We opened a designated mail called ”schedule_summoner@gmail.com”, and created the credentials and tokens for this email, so we can use Google Calendar API. After our algorithm output an assignment to the scheduled week, the user is able to press a button, and export the schedule into Google Calendar - the user gets an invite from ”schedule_summoner@gmail.com” to each of the relevant events, and then by accepting the invite, the user can easily add the events to their own calendar. An example of the output to google calendar: