

# University Exam Scheduler

Introduction To AI Final Project Report



Avinoam Hershler (208926964)

Alon Vaispaper(324817477)

Yair Maryles(313180713)

Dan Regev (208272518)

<b>Introduction:</b>	<b>2</b>
<b>Problem Definition</b>	<b>3</b>
<b>Solution Methods:</b>	<b>4</b>
Genetic Algorithm	4
Simulated Annealing	5
Evaluation Method	6
<b>Results and Comparisons:</b>	<b>8</b>
Genetic Algorithm:	8
Simulated Annealing:	9
Comparison with Human Solution:	11
<b>Conclusion</b>	<b>12</b>
<b>Appendix A - Data Formatting</b>	<b>13</b>
<b>Appendix B - Choosing Parameters for Simulated Annealing Algorithm:</b>	<b>15</b>
<b>Appendix C - Semester B solution analysis graphs:</b>	<b>17</b>
Genetic Algorithm:	17
Simulated Annealing:	18
Human Solution:	19
<b>Appendix D - Program usage</b>	<b>20</b>
Setup and Run	20
Input Files	20
Input Dates	21
Running a Solver	21
Displaying and Exporting Solution	22

## Introduction:

The problem we have decided to solve is how to set up and coordinate exam dates in an organized schedule. Apparently, in practice this is done in a manual method which requires consulting with each curriculum's representatives and taking into consideration the constraints that are made by their courses, e.g. making sure that compulsory courses do not have their exams on the same date. Furthermore, we would like to maximize the intervals between exams which makes this problem harder.

Our goal is two-fold:

- We wish to automate the long and tedious process of manually scheduling exams, that takes a tremendous yearly toll on student representatives who would be better off spending their precious time on their studies. In this regard our success will be measured by our ability to automatically provide a schedule for 2022 exams that is not worse than the current schedule.
- If possible, we want to provide a solution that is empirically better for the general student populace, according to the same standards defined by the representatives who work on the schedule.

## Problem Definition

The problem's input:

1. List of all Courses for each semester.
2. A weight for each pair of courses<sup>1</sup>.
3. A list of dates for each semester during which exams may be scheduled, for regular dates (mo'ed aleph) and additional dates (mo'ed bet).

The problem's output: an organized schedule of the exams with the following constraints:

1. Each course has two exam dates - a regular date and an additional date. There must be an interval of at least 21 days between those two dates. (Each student can take the course's exam on both dates. The first date is called Moed Alef, and the second is called Moed Bet)
2. Maximise the interval between any pair of courses according to the input weights. The exact method of weighting will be discussed in the section concerning evaluation.

Since there are dozens of curriculums and courses and a relatively small number of possible exam dates, the space of possible solutions is enormous.

---

<sup>1</sup> See appendix A for the explanation on the data we used, and how the weights are determined.

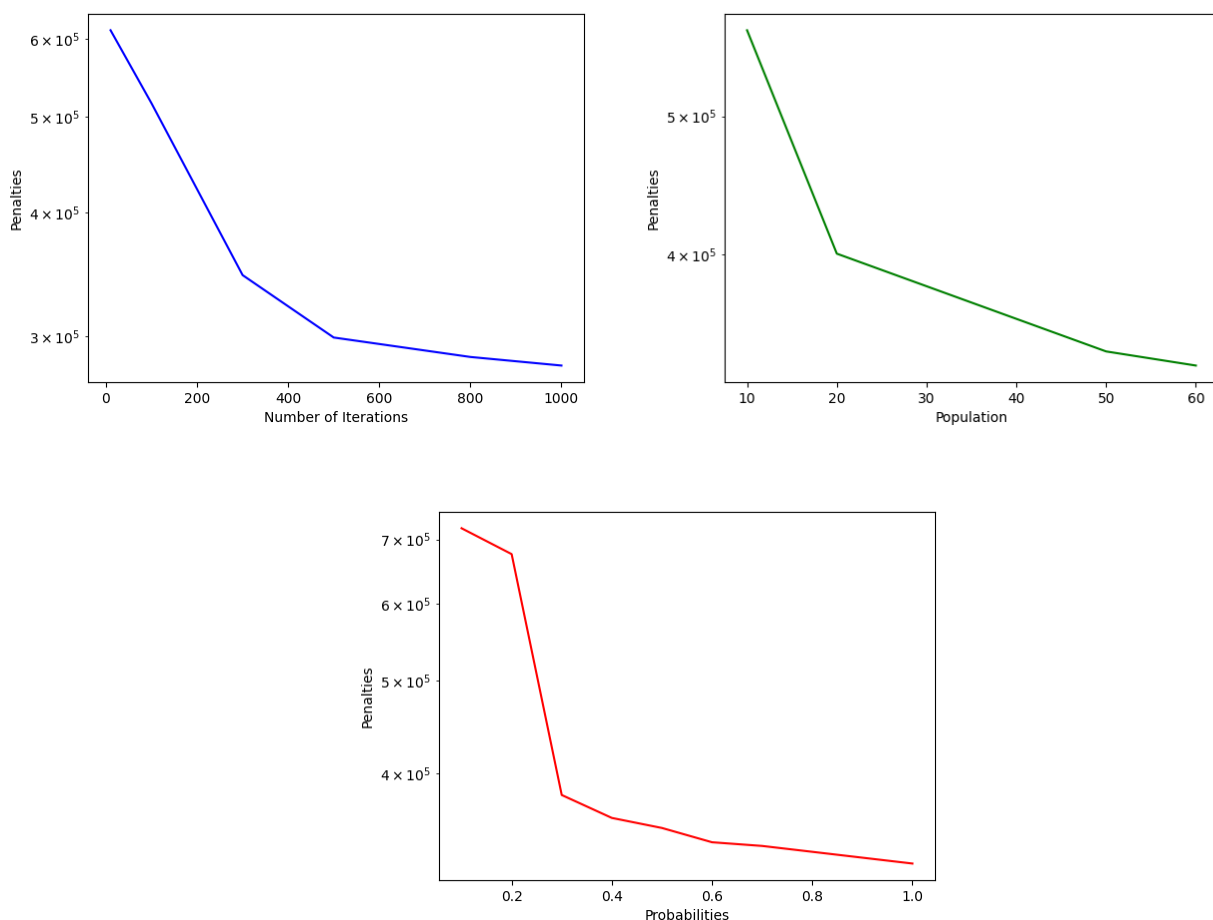
## Solution Methods:

Intuitively, the solution space for this problem is very large, and most solutions are legal, but may not be optimal (in fact, solutions are likely to be sub-optimal). To support this assumption, we decided not to consider solutions where compulsory courses are scheduled on the same day as illegal, but rather to evaluate them as bad solutions.

The result is that it is relatively easy to create a legal solution, and also to create new legal solutions from existing ones. This led us to attempt to solve the problem using local search - specifically Genetic local search and Hill-Climbing with Simulated Annealing.

### Genetic Algorithm

The Genetic algorithm is an approach inspired by the process of natural selection. We start with an initial population of randomly generated solutions, and in each iteration (A.K.A generation), we create a new population by probabilistically combining previous solutions and making small random mutations. We evaluate each solution using the evaluation function (details in the next sections) to determine its fitness (i.e probability of reproduction).



We defined a solution state to the problem as a list of pairs of dates corresponding to courses from the input. This means it is quite easy to combine two existing solutions by taking the first half of the date list from one parent solution and the other half from another parent solution.

Since both dates from each course were always taken from a single previous solution, the 21-day constraint is not broken.

We created random mutations in existing solutions by randomly selecting one course, randomly selecting one date (regular or additional), and setting it to a randomly selected date out of all the legal possibilities (that is, dates that are within the range and not closer than 21 days to the other exam date for this course).

The algorithm has some hyper-parameters: probability of mutation, initial population size, number of generations, etc. We tested our solver with various options until we found parameters that produced good solutions without running for too long.

## Simulated Annealing

Another solution to this problem is the Simulated Annealing solver. This solver works with a Simulated Annealing algorithm which iterates a certain amount of times and at each iteration tries to improve its current state. We built this project generically, so similarly to the Genetic solver, the SA solver uses states to represent some division of courses to test dates, and an evaluator to determine how good or bad some division is. During each iteration, a change is made to a state which might make the state better or worse, in terms of the evaluator. If a change is made which is an improvement from the old state before the change, then we always keep the change. Otherwise, with some likelihood we will keep the change even if the change decreases our value as a state, again and always, in terms of the evaluator. This is the most well known special aspect of Simulated Annealing and this is used in order to escape local minimas, which a regular hill climbing algorithm will get stuck at.

In addition, a common way to ensure that we reach the global minima and not just keep escaping local minimas, hoping to reach a local minima which is also a global minima, is to introduce a term called temperature. The Simulated Annealing algorithm actually gets its name from the annealing process in metallurgy, which in short is a technique of heating and controlled cooling of materials to purify its crystal elements. This is done by controlling the descent of temperature and the types of changes of energy throughout the temperature descent. Likewise, in order to raise the likelihood of finding the global minima, in Simulated Annealing we try to control the types of jumps that are made throughout the search by introducing this temperature aspect and determining delta (the likelihood of making a mistake to escape local maximas as explained above) with this temperature value. A mistake is made when a randomly unified number between zero to one is less than delta at a given iteration. We found that the most common way of using this method is by determining delta to be  $\delta = \exp((S_1 - S_2)/T)$  where T is our temperature value and S1, S2 are the two values that the evaluator gave on the state we had on some iteration before we made a change and after. In this way, as T decreases to 0, delta reduces to 0 as well and the probability of using the SA trick of making a mistake on purpose converges to zero as well. In this way as T, the temperature, decreases to 0, the SA algorithm becomes a regular hill climbing search. We implemented T to correspond to our iterations so that when we near the end of our iterations, T converges to 0 as well, which ensures that we end at a local minima. In order to ensure with a very high probability that this local minima is in fact the global minima, the delta shown above promotes bigger moves (bigger diff between S1 and S2) when T is large, when we want to search the search space, and promotes

smaller moves as T descends in order to randomly jump less and climb more, thus finishing at a local minima which is the global minima with a high probability.

We were not satisfied by this solution since even if we controlled the temperature and thus the jumps and search of the algorithm, we still didn't have a way to ensure that we will not move away from the global minima's mountain range once we've found it. We also didn't have a way to ensure that we would escape getting stuck in large drainage basins. (When a local minima is in a very big basin which is hard to get out of) In order to solve these two issues we introduced two more aspects to our SA solver. The first is very simple which is to keep track of the best results found and return to it after a certain amount of iterations in order not to "lose" the best position, before the hill climbing stage, if we already reached it. The second aspect we introduced was to have a few types of generators. (Functions that change S1 to S2 in different ways) With different types of generators we are able to change the "paths" we take throughout the topological search space. This can be easily explained with an example from the known Traveling Salesman's problem. In the case that we have a suboptimal solution A (1->3->2->5->4->7->6...->n) and an optimal solution B(1->2->3...->n), where each number is a city that TS must reach, we can see the importance of generators. While using a generator that swaps two numbers each time, solution A and solution B are very far away from each other. However, while using a generator that swaps k even and uneven cities with each other, these solutions are a lot closer together. An easy way to think about this is that the search space can be traveled along by foot but then we might reach a cliff or have to go a long way around a deep basin, however if we had ladders and bridges, then we can travel with more ease across the search space. We implemented this by choosing a few types of generators and therefore improved our quality of search in the search space. The generators we chose are on 1-3 courses, either swapping their tests dates with another course at random, moving both of their dates either a day back or forwards, or moving either their moed alef or moed beit date a day back or forwards. (as long as the date was a possible date to take a test on) In order to enforce the hill climbing ending to the algorithm, during this final stage we only allow this solver to change one date in one direction on either one or two courses and only keep that change if the state with the change is better than the old state in terms of the evaluator.

In all we built our SA solver to run on X1 number of iterations, with a linear reducing temperature, while our generator is changed every X2 number of iterations and we return to our best result found so far every X3 number of iterations until our last stage at X4 iterations where we conclude the search with a regular hill climbing search to reach the local maxima. In this way and with the delta function given we ensure that we reach the global maxima with a high probability.

Our choices for X1, X2, X3, and X4 are explained more in detail in Appendix B.

## Evaluation Method

In both of the methods described above, we used a State object as part of the algorithm. The state object serves as a possible solution to the problem - it contains a mapping of the courses to their exam dates. In order to perform the Local Search, we had to evaluate each state. The evaluation can be thought of as a penalty we give to the solution - the lower the evaluation - the better.

In order to give a penalty, for all possible pairs of courses, we assign a certain "weight" - a number which signifies how far their exams should be scheduled. The weights are calculated from

the data provided to the program, before running the algorithms. Each course's pair has 7 possible collisions between them in some major's semester:

- They are both "HOVA".
- They are both "HOVAT BHIRA".
- They are both "BHIRA".
- One is "HOVA" and the other is "HOVAT BHIRA".
- One is "HOVA" and the other is "BHIRA".
- One is "HOVAT BHIRA" and the other is "BHIRA".
- There is no collision between them.

Each type of such a collision has a certain weight - a number which represents how relatively big the interval between the exams of two courses should be.

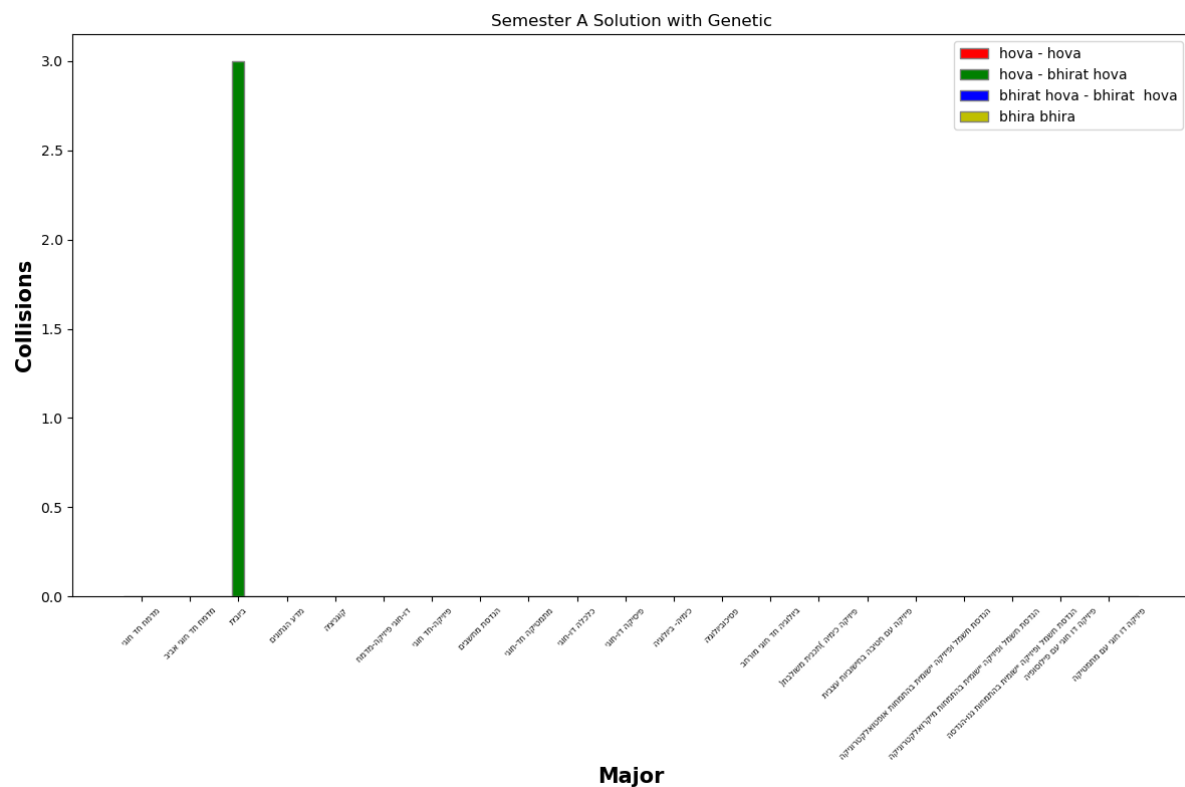
So, when we want to evaluate some state, we look at all of the possible pairs of courses. For each pair, we know how far their exams should be scheduled relative to others. If the weight of a certain pair is 'w', and the distance between their exams is 'd' days, we added to the penalty of the state:  $w * (1 / d)$ . This is done for each pair - its moed A and moed B.

It is worth noting that this is only one example of a way to give a penalty to a state. An interested user can easily write code for his own evaluator.

We note that the average interval between compulsory courses per major is centered around 8 days.



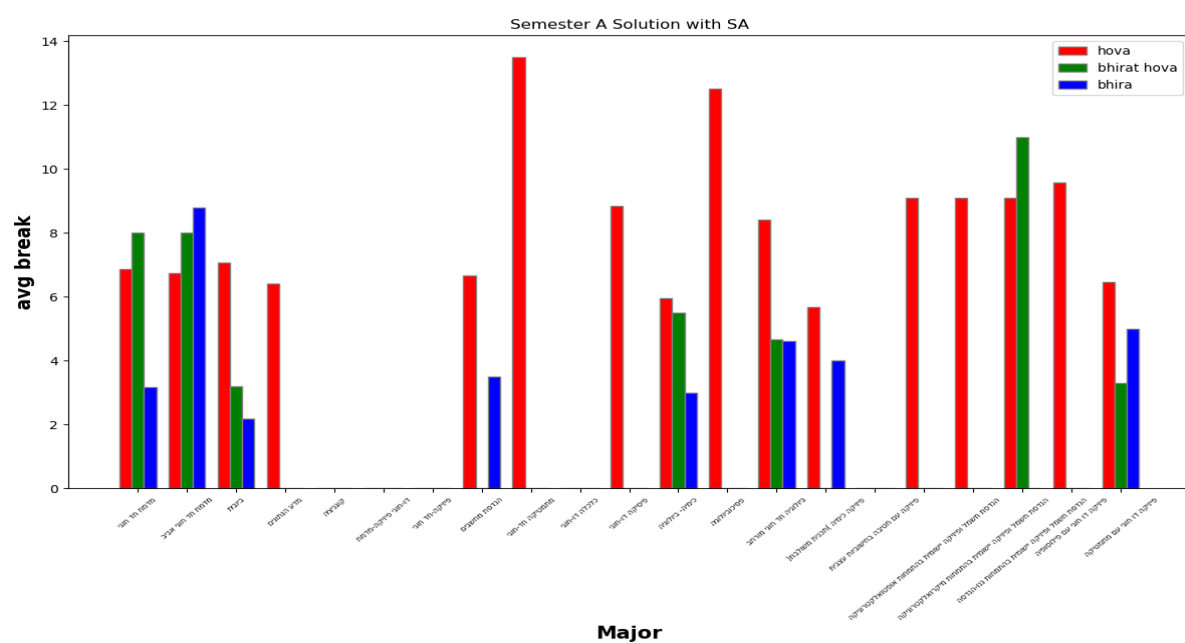
Number of courses scheduled on the same day:



We can see that there are almost no collisions in any of the majors. This is very good, and indicates that our algorithm found a good solution. It confirms the validity of our original assumption, considering collisions as a soft constraint that does not invalidate a solution state.

### Simulated Annealing:

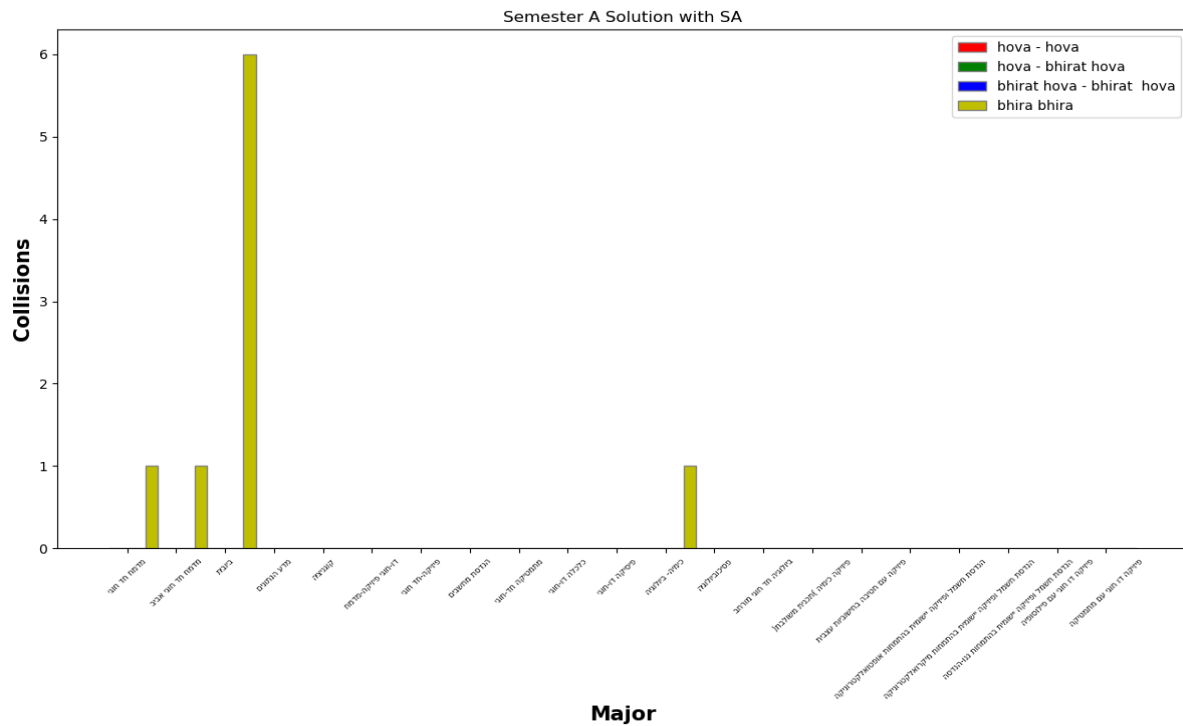
Average number of days between certain courses types in major:



Here we can see yet again, that the intervals between HOVA courses is generally higher than the intervals between the BHIRA courses, which is what we expect to happen.

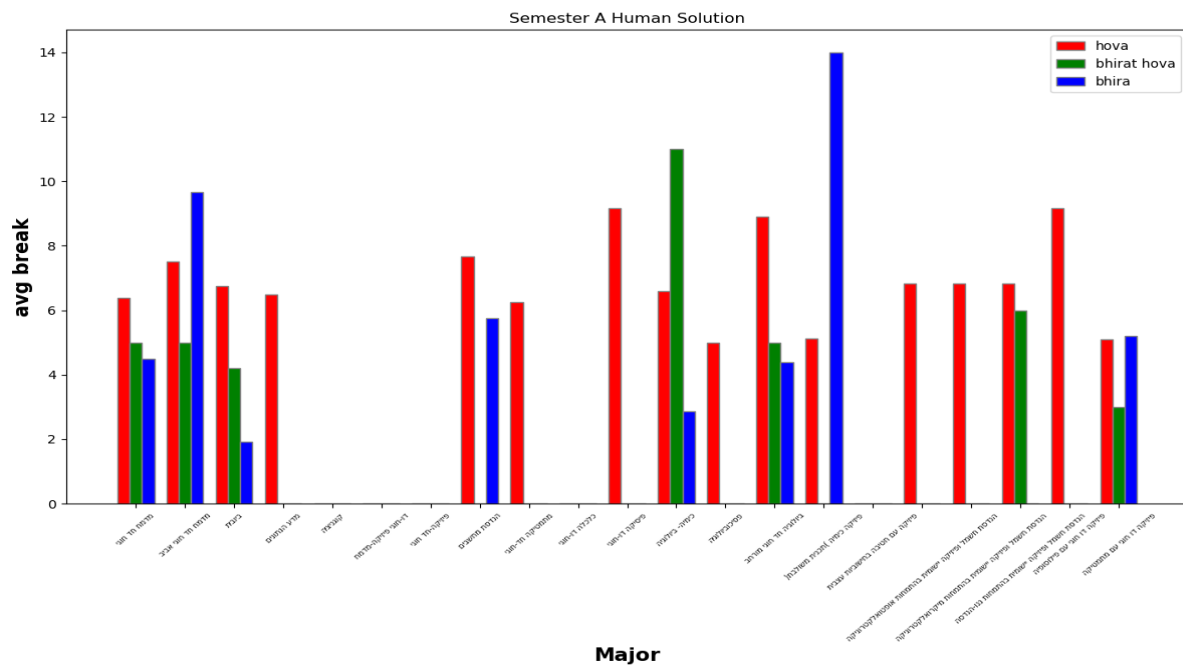
Once again, we note that the average interval between compulsory courses per major is centered around 8 days (with more outliers in this case compared to the genetic solution).

Number of collision:



We can see that most of the collisions happen in the BHIRA courses, which makes sense since they are less important, and there are many courses of this type.

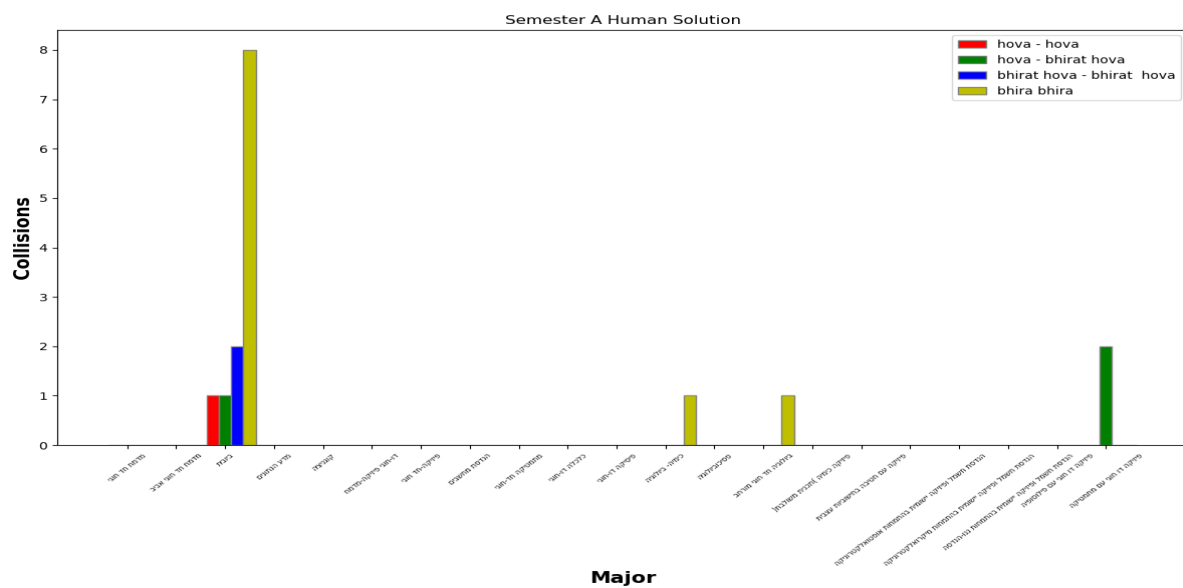
Average number of days between certain courses types in major:



We can see that generally we have bigger intervals in the HOVA courses (but not always), and sometimes relatively big intervals between the BHIRA and BHIRAT HOVA courses which are less important. This may indicate that a better solution could be found.

We also note that the average interval between compulsory courses is mostly under 8 days, which is worse than both of our solutions.

Number of collision:



We can see that there are collisions in some BHIRA courses which is ok. However there are also collisions in HOVA courses which is bad. We can see that this human solution is not good.

## Conclusion

Although the only true empirical test for the optimality of an exam schedule is having 20,000 students from all the various majors actually take the exams according to the proposed schedule, we feel confident in concluding that the results produced by our algorithm are not worse than the existing solution, thereby fulfilling our first goal.

While this claim is more difficult to empirically justify, we can see that with regard to the empirical parameters we defined, our solutions are better than the current solution.

It is also worth mentioning that close inspection of our solutions by the student representatives - who collectively represent the world's foremost expertise on exam-schedule evaluation - may lead to some fine tuning of the optimality parameters defining our evaluator. The way we designed our project leaves us confident that this kind of tuning can be performed easily and can lead to good results that are tailored toward the priorities of the students.

We feel that our second goal has, therefore, been reached..

## Appendix A - Data Formatting

### First attempt:

\_\_\_\_\_ Initially we thought of taking data from previous years. Specifically we wanted to get the list of students studied in each course. Then, according to that information, for each pair of courses we could have determined how far they should be scheduled one from the other according to the number of students that studied in both of the courses.

Unfortunately, the university didn't agree to give us this data, so we had to think of something else.

### Second attempt:


\_\_\_\_\_ Failing to get any information from the university administration, we decided to create a google spreadsheet, and asked the students committee of different majors (מסלולים) to fill in the data of their majors.

We realized that in most (if not all) of the different majors, there is a well defined structure: each major has at most 4 years of study (which are 8 semesters), and in each semester there are "HOVA", "BHIRAT HOVA", "BHIRA" types of courses.

Our second attempt (which ended up being the one we used), is working as follows: given a pair of courses, c1 and c2, we would like to give this pair a score which indicates how far away their exams should be scheduled from one to another. We are doing it by searching if there is any common major between these two courses:

- If the courses are not taught in the same semester in the common major, we can conclude that their exams can even take place on the same date - there should be any students studying this major, who will take both of these courses at the same time.
- If the courses share the same semester in some major, their distance from each other in the schedule is dependent on their types: if, for example, c1 and c2 are both "HOVA" in the given major, then we should make as much space between them as possible. If, however, these two courses are "BHIRA" - the time period between their exams can be relatively small.

This is the main data we used in our project. Here is a link to the spreadsheet:

 [איסוף מידע על קורסים לפי מסלולים](#)

In the submitted files, this file is simply called "major\_data".

However, this data was not enough. For example - we don't have the courses names, and we can't know if these courses have exams (there are courses, like Intro to AI, that don't have an exam).

In order to solve that, in addition to the data described above, we also had a list of courses, together with their names, which had an exam in “Givat Ram”.

This file was created by the committees of the different study majors in “Givat Ram”, in which they tried to schedule dates for the exams next year by hand. So, we used this file in two ways:

- Via this file, we can now have the list of courses together with their names which have an exam.
- In this file, there is also a human solution to the very same problem we are trying to solve - a great way to test our results against human solutions.

So, we now have two additional types of files:

- `courses_names_A (or B).csv` : A list of semester A (or B) courses numbers and their names which have an exam.
- `sem_A(or B)_sol_human.csv` : A file containing a representation of the human solution to our problem for semester A (or B).

In Code Data Representation:

First, we constructed two main objects to hold our data: A Course object, and a Major object. These two objects interact with each other. A course object will have its rough data (like the course’s number, name), and the majors the course is being taught at, including their semesters in each major and types. The Major objects simply contain the schedule of the major - list of courses for each semester and their types.

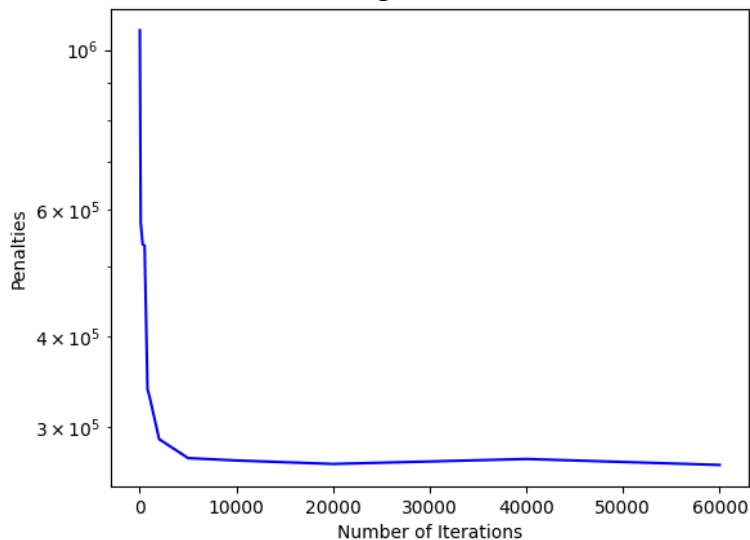
Now, notice that solving the exams schedule for semester A and semester B are two different tasks. With that being said, every time we read the data, we can only read data of semester A or semester 2. Therefore, the object that reads the data - CSVdataLoader, reads the data of on of the semesters (A or B) at a time. This is enough for the solution for the current semester (A or B).

## Appendix B - Choosing Parameters for Simulated Annealing Algorithm:

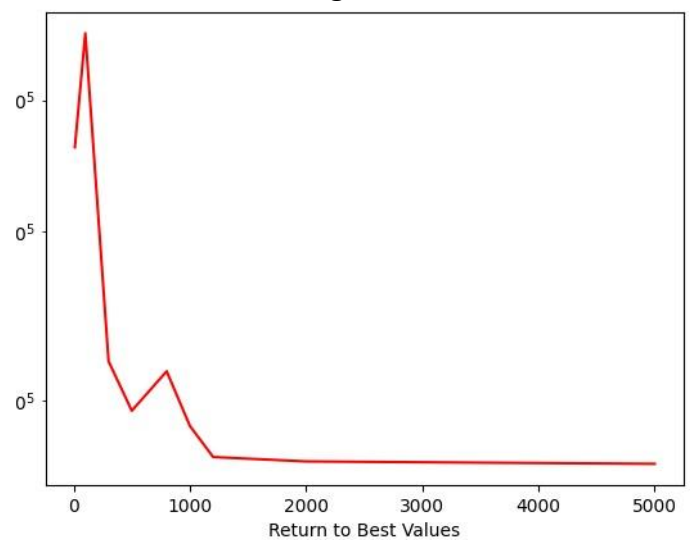
In order to optimally run the Simulated Annealing algorithm we had to find the optimal parameters that the algorithm runs with. As explained in the SA section, the four parameters we had to find are the amount of iterations to run the SA solver on, the amount of iterations that we want to re-generate a new generator, the amount of iterations that we want to re-spawn at the best position found so far, and the amount of those iterations that we want to save for the last stage,.

First we found the amount of iterations we wanted our algorithm to run on. Figure 1 shows how the algorithm converges already after 5'000 steps and only improves slightly with more iterations. Since runtime is an important factor, we decided to run the SA solver on 7,000 iterations which leaves us extra room to constantly return a good result. We then search for the best value for returning to the best state found so far, in order to not “lose” our best local minima once found. As seen in Figure 2, this value converges at around 1,500. We can also see that when this value is too low, we don't allow the algorithm enough iterations to actually determine if it's in a good or bad mountain range, thus not letting it have enough time to reach local minimas.

**Figure 1**



**Figure 2**

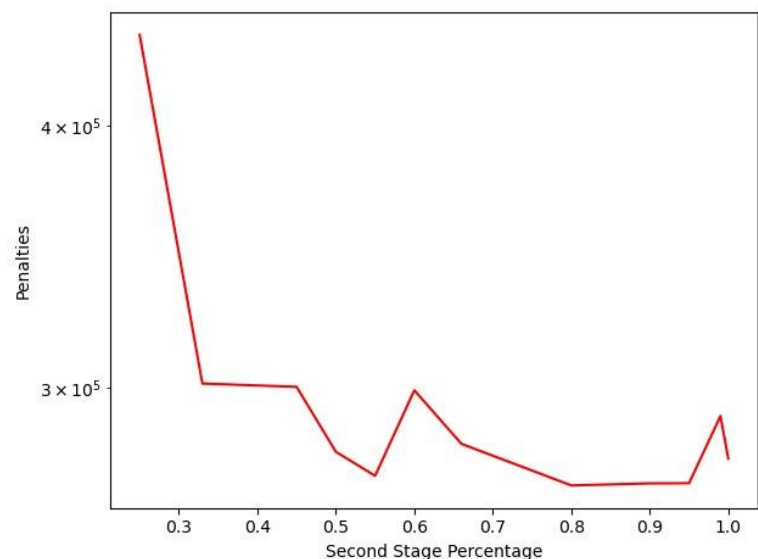
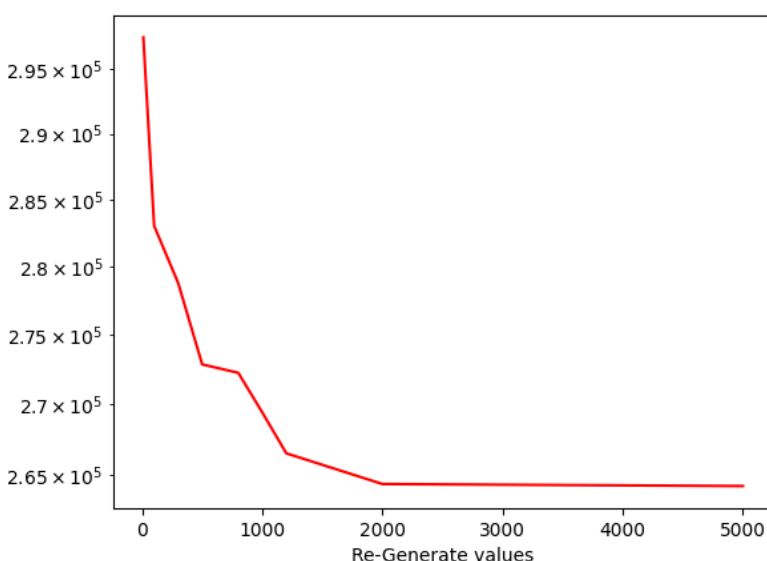


We then looked for the optimal amount of iterations that we want to re-generate a new generator in order to escape large drainage basins. Figure 3 shows that the re-generating parameter converges at around 2000 iterations, therefore our algorithm re-generates every 2,000 iterations to ensure that we escape large drainage basins by re-generating the most amount of times while also giving the algorithm enough iterations with each generator type to make a real difference. We can see in the figure that when this value is too low, the generator is switched to quickly in order to make any real improvement.

Lastly we found the best value to enter the last stage, a.k.a after how many iterations we enter the final hill-climbing stage. This is important on two fronts, on one hand it is important to leave iterations to hill climb in the final stage in the case that the global minima's mountain range is found towards the end of searching the search space, and on the other hand it is important to leave many iterations for the transition phase between mostly searching and mostly hill climbing. This was controlled by a parameter of the percentage to leave for the transition phase, which directly affects the iteration number where we move on to the last stage. Figure 4 shows how the percentage of the transition phase, or the second stage, has a very big effect on the solver. We can see how when the transition phase is small this algorithm doesn't work well since it jumps around a lot in the first stage, does a quick transition phase, and then works mostly with the last stage, which is the hill climbing phase, leaving the algorithm to finish most likely at a local minima and not the global minima. In addition when the transition phase is very large, we can see that the algorithm can return a suboptimal solution since we didn't leave iterations left for regular hill climbing to ensure us that we are left with a local minima at the end of the algorithm's run. Therefore we took 80% as the percentage for the second stage, the transition phase, leaving us with 20% for the first and last stages to search around the search space at the beginning and to hill climb at the end to reach a local minima which is most probably the global minima.

**Figure 3**

**Figure 4**

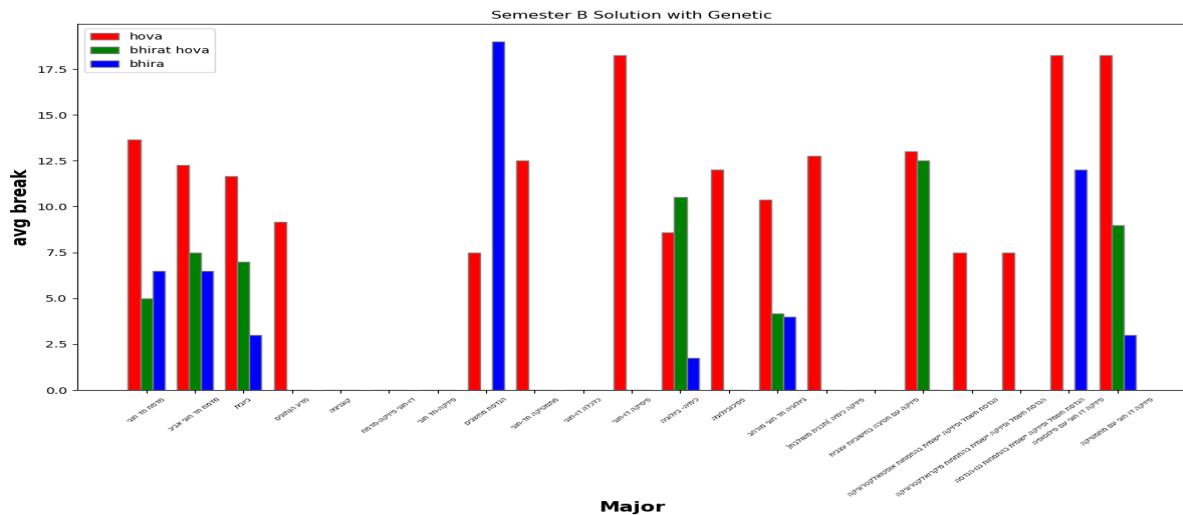




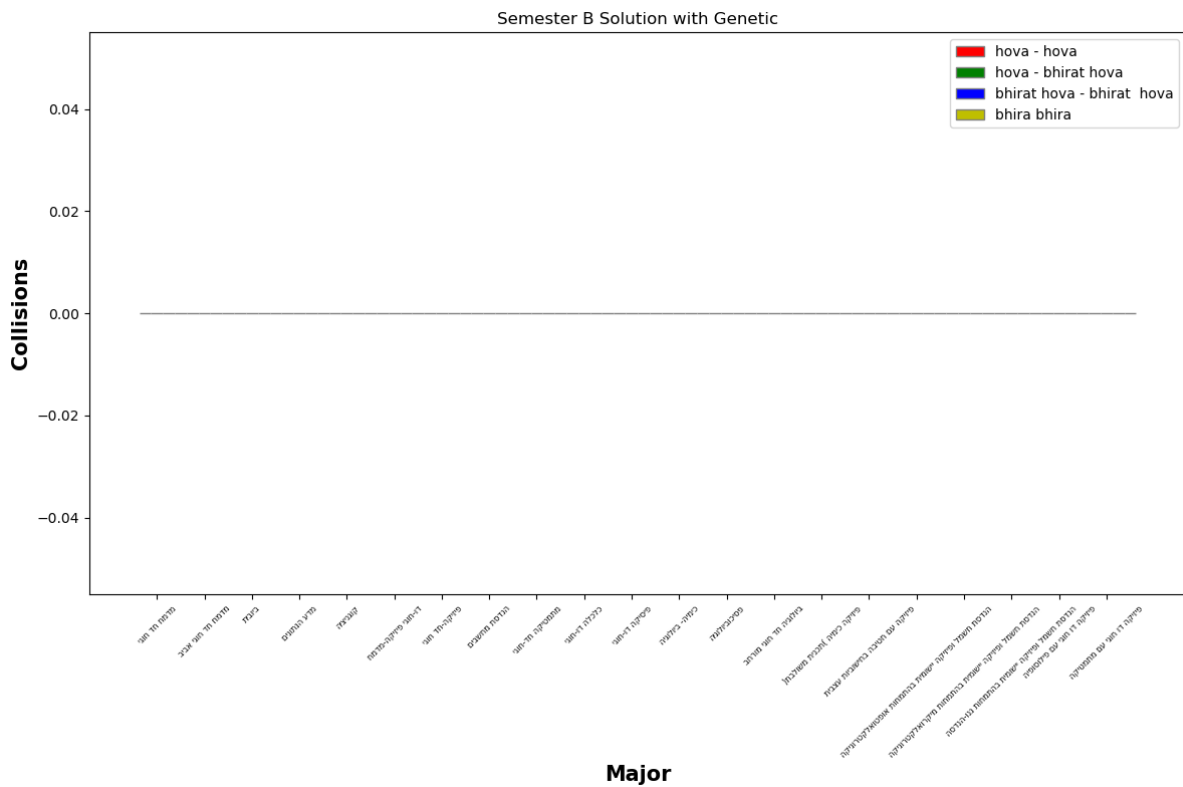
## Appendix C - Semester B solution analysis graphs:

### Genetic Algorithm:

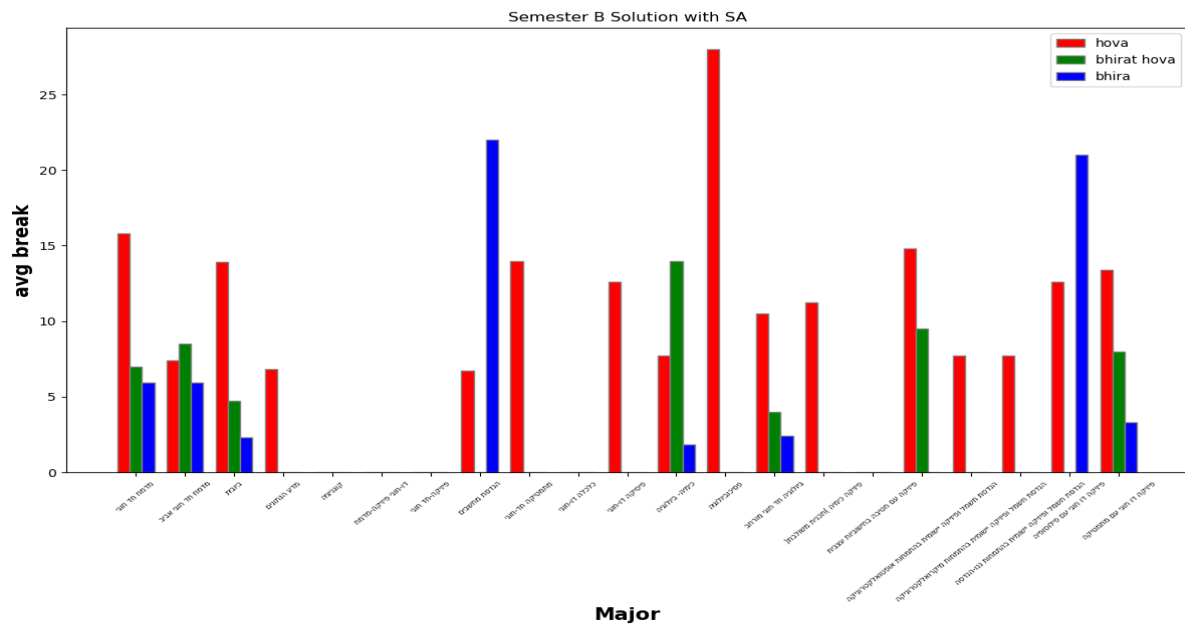
Average number of days between certain courses types in major:



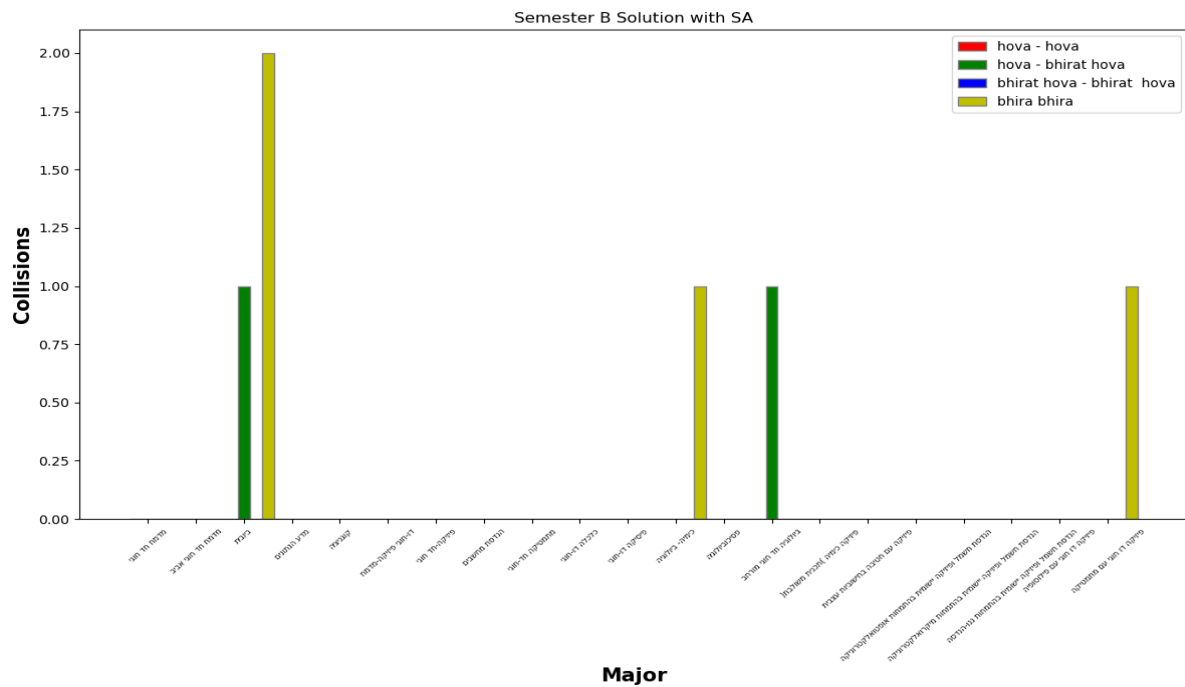
Number of collision:



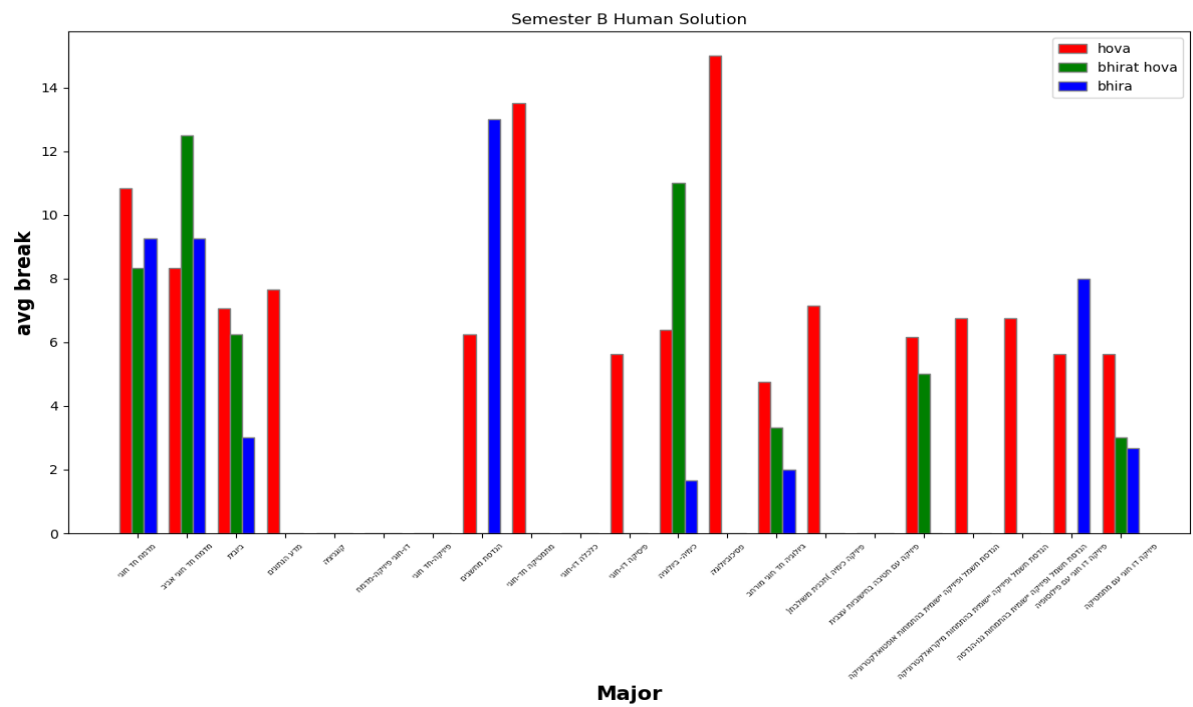
Average number of days between certain courses types in major:



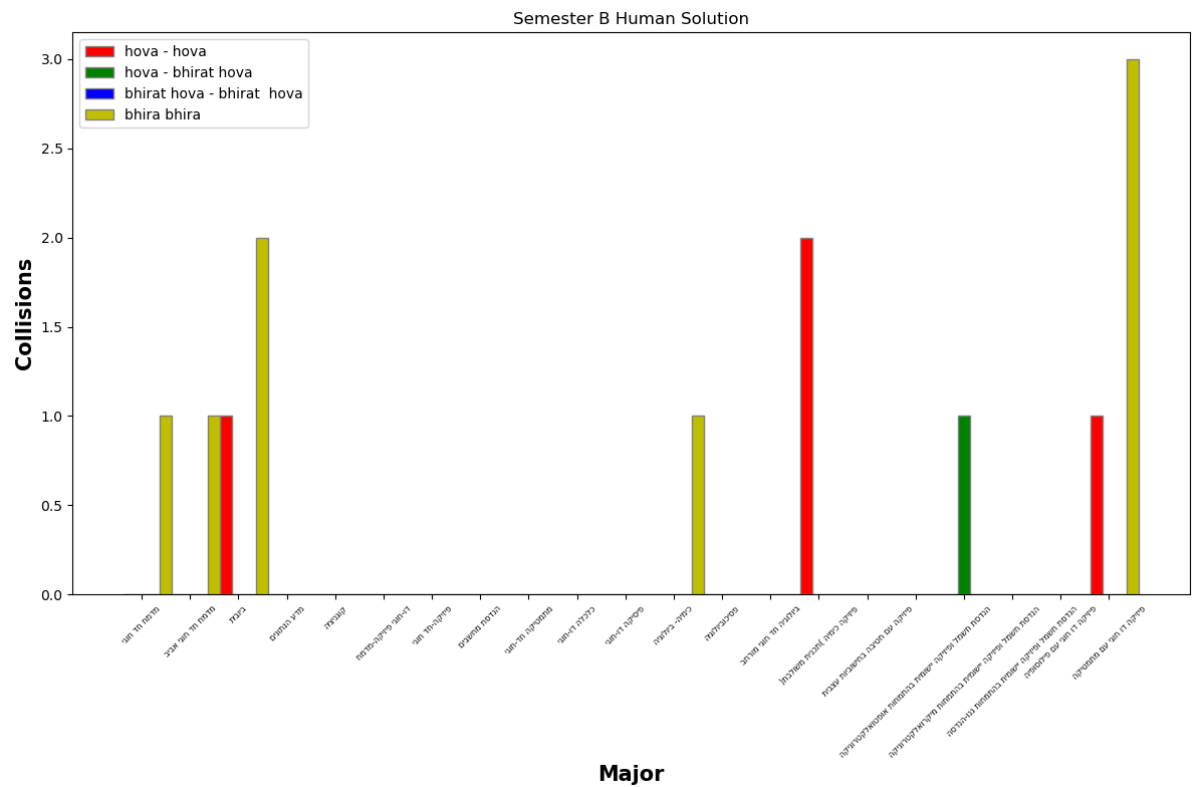
Number of collision:



Average number of days between certain courses types in major:



Number of collisions:



## Appendix D - Program usage

### Setup and Run

To run our project, you must first provide a python environment supporting our requirements.

You may do this easily using the requirements.txt file we provide, for example:

```
python3 -m venv my-virtual-environment
source my-virtual-environment/bin/activate
python3 -m pip install -r requirements.txt
```

Once the environment is set up, you can run the program's GUI by executing:

```
python3 ExamScheduler.py
```

A window will open looking something like this:

On the left is a panel for entering all the information for the problem input, and on the right there is a calendar display.

**Exam Scheduler**

**Setup**

Load Data

Majors Data File

Semester A Courses File

Semester B Courses File

Setup Schedule

Semester A Exams start

Semester A 1st round end

Semester A 2nd round start

Semester A 2nd round end

Semester B Exams start

Semester B 1st round end

Semester B 2nd round start

Semester B 2nd round end

Solver Algorithm

Number of Iterations (0 for default)

**Display Calendar**

September 2021

Sun	Mon	Tue	Wed	Thu	Fri	Sat
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

## Input Files

The program requires three input files:

- Majors Data File: A csv file containing the information about each major (see appendix A).
- Semester A Courses File: A csv file containing the names and numbers of courses that have exams in semester A (see appendix A).
- Semester B Courses File: A csv file containing the names and numbers of courses that have exams in semester B (see appendix A).

Relevant files for 2022 are provided in the data folder:

- Majors Data File is called: `majors_data.csv`
- Semester A Courses File is called: `courses_names_A.csv`
- Semester B Courses File is called: `courses_names_B.csv`
- 

## Input Dates

For each semester, The program requires four dates:

- Start of regular dates (mo'ed aleph).
- End of regular dates (mo'ed aleph).
- Start of additional dates (mo'ed bet).
- End of additional dates (mo'ed bet).

The ranges we used for our 2022 solution are:

- Semester A: 16/01/2022 - 11/02/2022, 13/02/2022 - 04/03/2022
- Semester B: 26/06/2022 - 27/07/2022, 28/07/2022 - 19/08/2022

These dates may be input using the date entry widgets, or through right-clicking on the calendar display. Once they are selected, the ranges will light up on the calendar display.

Specific dates may be defined as days with no exams (in the case of national holidays, etc.) using the right-click menu on the calendar.

## Running a Solver

The program provides a choice between the two implemented solvers: Genetic and Simulated Annealing.

Clicking 'Solve' will run the solver for both semesters for a default number of iterations. This may take a long time to complete (a bar will be displayed to indicate progress). Optionally, the user may input a specific amount of iterations for the solver to run.

## Displaying and Exporting Solution

Once the solver finishes, the screen will display something like this:

The scheduled exams are shown on the calendar display.

On the left there are some controls for viewing the solution:

- A checklist allows filtering the solution to view exams relevant to specific majors and semesters.
- Two buttons allow the user to jump quickly between semester A and semester B dates on the calendar display.
- Two buttons (one for each semester) allow the user to save the provided schedule to a csv file.
- A checkbox that defines whether to show course numbers or course names on the calendar display.
- A reset button allows the user to return to the first screen and run a solver again.

The screenshot displays the 'Exam Scheduler' application window, which is divided into two main sections: 'Setup' on the left and 'Display Calendar' on the right.

**Setup Section:**

- Calendar view jump:** Includes buttons for 'Jump to Semester A' and 'Jump to Semester B'.
- Show Course Names:** A checkbox that is currently unchecked.
- Save Schedule:** A button to save the current schedule.
- Save Semester A Schedule:** A button to save the schedule for Semester A.
- Save Semester B Schedule:** A button to save the schedule for Semester B.
- Reset:** A button to reset the application.

**Display Calendar Section:**

The calendar displays a grid of dates for the month of September 2021. The days of the week are listed at the top: Sun, Mon, Tue, Wed, Thu, Fri, Sat. The dates are arranged in a grid, with each date cell containing a list of course numbers. For example, on September 1st, the courses listed are 72180-1, 77101-1, 83312-1, 80415-1, and 76631-1. The calendar also shows the start and end of Semester A (September 1st to September 30th) and Semester B (October 1st to October 31st).