AI Project - Final Report
‘Tama 48’

19 August 2018

Adi Yehezkeli 300203791
Naama Glauber 312513583
Ariel Bruce 332668854
Rony Ginosar 203810221

***execution note*:** install Flask, then run from the command line “python server.py” under the TAMA48 folder. The commandline should show: “ Running on http://0.0.0.0:8080/ Restarting with stat” . Open your browser and paste the url that is written in your command line. Your browser should do the rest :)

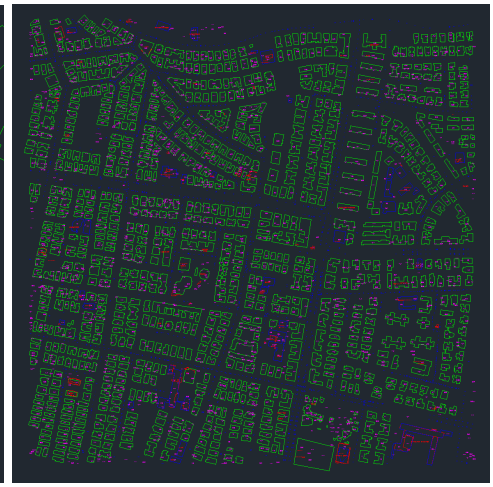
What problem are you going to solve?

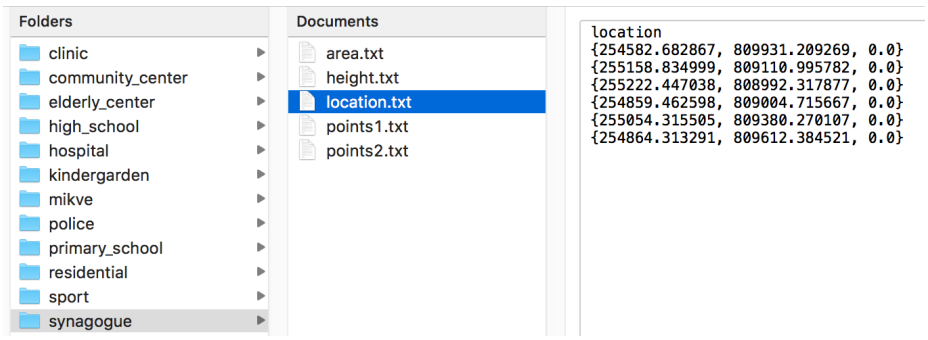
In the next 30 years, by 2048, the population of Israel is projected to double, from 8.5M to 15M. Land is a finite resource; while new towns and cities can be developed, the vast majority of citizens will live in already existing cities. Therefore, new residences must conform to the constraints that result from the geography of pre-existing towns and cities. Our goal is to develop an intelligent solution that will allow to add population to inhabit a specific neighborhood, while satisfying the social, environmental, and economic constraints, and maximizing the benefits from existing public buildings.

How are you going to solve it?

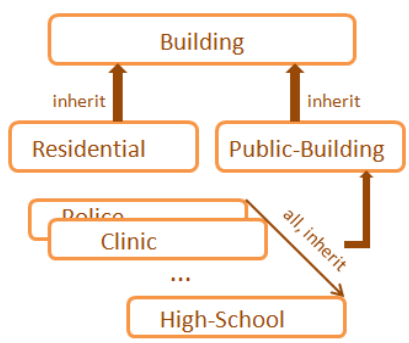
The project includes four main steps:

1. **Data Preparation:** in order to accommodate the new population, we need to understand the current state of a city. We used a CAD file of Tel-Aviv’s urban plan, for this purpose. To process the data, we cleaned the DWG file, created 3D rhino file, and extracted the buildings’ properties (coordinates, height, area and polygon) to files. We also labeled the public buildings according to their specified type as defined in the GIS map. Each type of public building is saved to a different text file, where each row belongs to a building, with an ID that is the row number.





2. Data Structure & Defining 'Solution':



A *building-type* is a type of public building, such as a police station, health clinic, or high-school, as well as residential buildings. Each object of *Building* includes all relevant fields (extracted from the files' data).

As can be seen in the diagram above, we ordered the data structure such that extending to future types of public building will be convenient. In the code, the data was read from the files and stored as List[Pair(building-type, List [Buildings-from-type])].

A solution is the same list of initial buildings data, updated with the extra floors (int) for each building object, both residential and public. We also return a score double value for the plan.

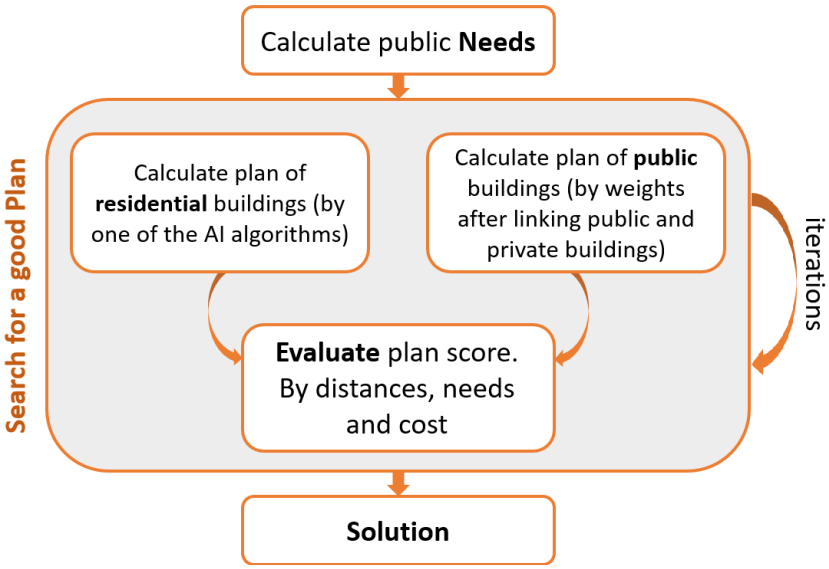
3. AI Algorithms:

According to known parameters in the field of architecture and urban planning and given the number of people we would like to add to the neighbourhood we developed an automatic solution to decide how many floors to add for each public and private buildings. The challenge is to find a vector that will minimize the cost and maximize the social benefits and needs.

We calculated an estimation for the needs of each type of public building, i.e. the number of required units to add for the population size we want to add. This estimation was based on known parameters for building size requirements followed by a government document called 'A guide for allocating public buildings' (picture above). For each type of building, the required area per unit is given (like class, or kindergarden) or required area per populations (like what size of a police station should be as a function of the population in the city). Some parameters like percentage of religious people, or percentage of children etc.. could be user's decision, but since we took a specific area, and for simplicity, in our project we fixed it to known standart.

Moreover, we linked between the residential buildings and the public buildings they are using. Therefore, each residential building has a list of public buildings it uses (one building from each type), and each public building has a list of all residential building using it.

The flow of our work includes iteration, while in each we have two levels of calculation and an evaluation of the solution (the order between public and residential calculations is different between the algorithms).



3.1. Calculate Residential Buildings' Extension:

3.1.1. Genetic Algorithm (Artificial Intelligence Algorithm #1)

The objective is to maximize the score evaluation (where 'score' is the quality value we described later in section 3.3). We start with *k* random solutions, named them 'states', of the residential buildings' additional floors. At each iteration we choose the best 1/4 states, and randomly merge them into new set of *k* states. In order to avoid local maxima, with a small probability *mutProbability* a random state will be created instead of that merge. The merge of 2 parents is done by iterating over the additional floors vector of both parent, and for each building randomly choosing the value of the first or the second parent. Then if the amount of housing units added does not match the amount needed, it keeps randomly selecting buildings and now change the value to the min/max among both parents, until the right amount of units is restored. In order to improve the results, we made some small changes to the basic algorithm:

I. We always keep the best states and not only their children, so that the result score will never decrease.
II. When reproducing, we created from each pair of parents 4 times the amount of needed children, and selected the best ¼ out of all the results. This way we were able converge faster.

As a result of these changes- for each iteration, if the score is higher than the result score in the previous iteration, then one of the 4k new states is better than all ~4k * (iters-1) states examined before.

Analysis:

For this algorithm we should fix two parameters, *mutProbability and k*. Of course running our algorithm with *#iterations -> inf* and *k -> inf*, can find better results, but this is equivalent to running all the options. Ideally, we want to assign values for these parameters so it will converge to a good solution (higher score), after a short period of time (time is a function of number of iteration and k).

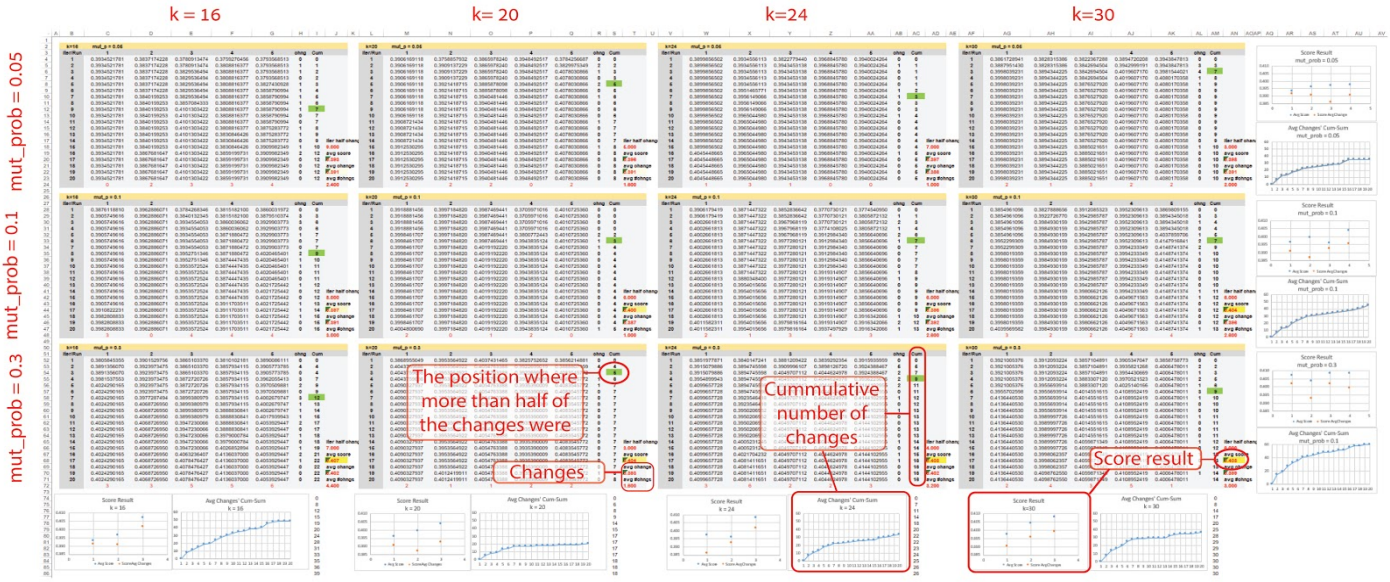
For that, three tests were executed:

I. Convergence: we found out that around 30 (sometimes 20) iterations we reach to convergence and the result usually will not be changed much later. To ensure that, we run several tests with 100 iterations. For example using k=[16,30], mut_prob=0.3 for 10k additional units, we found out that for k=16 around the first 30-35 iterations we got an average change of 0.12, while in the rest 65-70 iterations we got an average change of 0.02. And for k=30 in our runs, we got no change after 30 iterations!

   Moreover, we saw that too early convergence (to low values) happened more when we used low k (do not provide a lot of options), and low *mutProbability* (prevent from searching farther). It caused to local maximization. We tried that with *mutProbability*=0.01 with *k=8*, 16

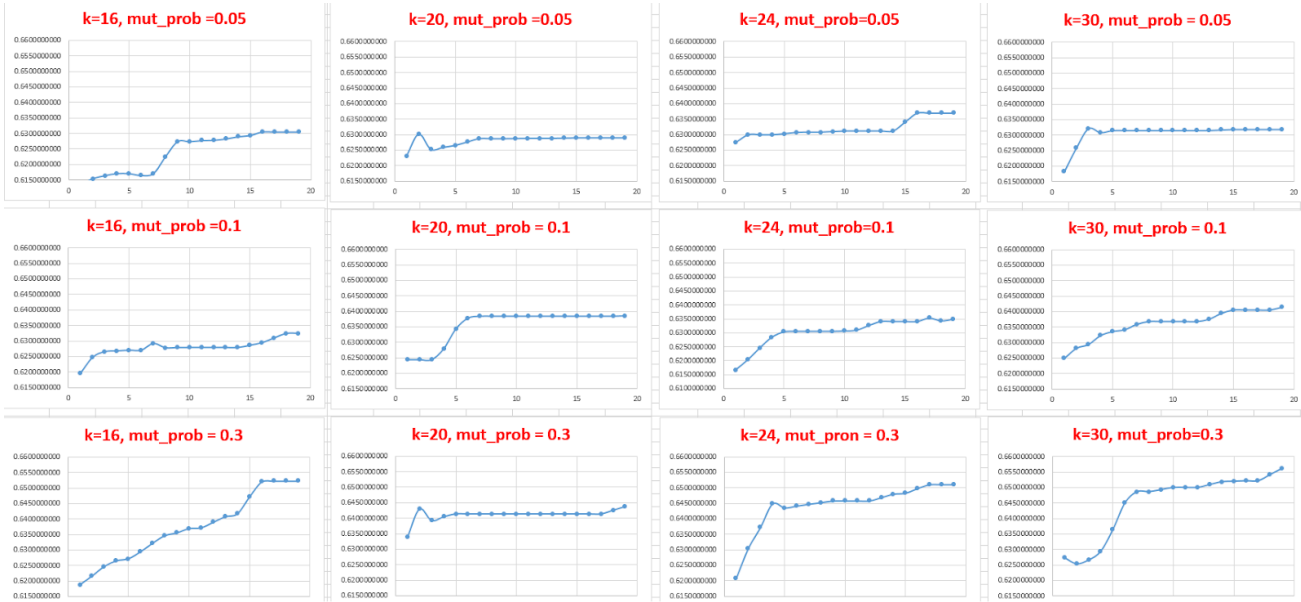
II. Random vs. AI genetic algorithm: The parameter *mutProbability* is telling us how much random we use versus using combinations of the best results from the previous iterations. When *mutProbability* is low value like 0.01, we don't explore much beyond the combinations of the previous results, so we might be stuck in local maxima, but when *mutProbability* is very high like 0.99 it is very close to check only random solutions. Therefore we need some balance here.

III. Fixing the parameters: Since a user do NOT know and probably don't want to deal with setting this parameters, we fixed it in the code, although easily it could be changed. We tried a few combinations of the parameters- *k*=[16,20,24,30], *mutProbability*=[0.05,0.1,0.3], #*Units*=[1,000,10,000], during 20 iterations, and and we run the algorithm five times for each combinations to reduce the noise of the random. We analysed the results separately for different number of units (because it is not comparable, there are different needs and the score is relative).



Here above, is the results for 10K units, all 24 combinations of k and *mutProbability*. The graphs on the rights are averaging each row, and on the bottom they average each column. The motivation was to look at the score, the changes between iterations (how many, how much, when converging), while fixing one parameter and checking the other.

We plotted the changes of the score between one iteration to the previous iteration (in average over all the five tests) to see more about the convergence.



Conclusions:

I. We assigned k to be 30: Less than that- we won't have enough options to produce from the current state, and more iteration will be required for nice results.

II. We assigned the number of iterations to be 30: Usually not many changes happened later (for *mutProbability* =0.3), and those changes are not very important ones, as described in the first test.

III. We assigned *mutProbability* to be 0.3: For this value, we decisively got better solutions, we wanted to check if increasing it won't give us even better solutions (0.3 means that 70% of the states are from merging, while only 30% of it is random, if it is 1.0 it means that it is a 100% random). And this is why we check later for 0.5, 0.7 and 0.99, to see that increasing it not necessarily provides better results.

For example, for 10,000 additional units we got this average result:

0.05	**0.1**	**0.3**	**0.5**	**0.7**	**0.99**
0.3905	0.3957	**0.4067**	0.4062	0.4007	0.3917

More:

- If run time time is an issue then k can be decrease to 20 instead of 30.
- When decreasing k it is better to increase the number of iterations.
- *mut_probability* can be decrease to minimum 0.2 instead of 0.3, and maximum 0.5.
- When decreasing *mut_probability* it is better to increase k (for better results).

3.1.2. Min-Conflict (Artificial Intelligence Algorithm #2)

In this algorithm, we build a solution step by step by increasing each time just one building, comparing to the previous solution of genetic Algorithm, where we started with a full random solution and manipulating it at each iteration.

In this algorithm we wish to minimize the amount of conflicts, while here the conflicts are calculated in the following way: since a strict true/false conflict does not exist for this kind of problem (it's not that a child will not have any school to attend, but rather will have a bad or far away one), we chose to use a concept of *conflict-points*. For example, if a kindergarten has 40 children per class, while the ideal class size is 27, each additional child is considered as a *conflict point*. The idea behind this kind of calculation is that there is a series of logical statements "a class has no more than 27 children", "a class has no more than 28 children", "a class has no more than 29 children"... etc.

The min-conflict algorithm works as follows: we start with only the original state of buildings, and in each iteration we pick the residential building with least conflicts. Then we raise it by one floor, count the housing units added, and re-calculate the public building's needs, now that the population grew. If additional public area is needed we add it (the same way we did in the genetic algorithm, as described later in section 3.2), and continue to the next round, until all the residential unit's needs are filled.

Analysis and Conclusion:

We run the algorithm for different number of units (This algorithm is deterministic, so every run with the same amount of units will have the same result). In general, the algorithm performed very well on a small amount of units, and deteriorate when the amount of units starts to grow. On a large amount of units, the algorithm "stucks" in a local maxima, because there is no way for it to go back and change what already been decided, that sometimes the greedy solution, after a lot of additions turns out to be wrong.

The idea of conflicts is not very natural for this kind of problem, since in the real world quality of life is not a true / false formula. This is why we had to be more creative in this part and find a way to describe the conflicts as true / false statements, and still it feels a bit artificial., but still this method works very well when only a small amount of units needs to be placed - because the algorithm "scans" the whole state and is able to decide where are the best spots of the city.

3.1.3. Min-Conflict vs Genetic:

Unit / Method	**Random**	**Min-Conflict**	**Genetic**
100	**0.2928**	**0.4809**	0.3103
1,000	**0.2681**	0.2891	**0.3049**
10,000	0.3675	**0.1597**	**0.4082**

As we can see in the chart above, Min conflict algorithm is significantly better for small amount of units, while the Genetic algorithm is better for big amounts, and therefore they complete each other. This happens because for a small number of units, the greedy Min-Conflict algorithm scans all the possibilities and picks the best places to add more units. Because it's not a lot of units, their effect on the state is small, because to begin with it picked the spots where there are "extras" of public resources. When the amount of units starts to be significant, their effect is big, and a lot of new public spaces have to be added. Now they will add up where all the new population was placed, which is not necessarily the best configuration.

In the Genetic algorithm on the other hand, the placement of a small amount of units is very random, since only a small amount of additions is spread between all the buildings, and this makes it very hard for the algorithm to converge. On a big amount of units though, the problem becomes less discrete and more nuanced, and combinations of states can more easily create better states.

3.2. <u>Calculate Public buildings' Extension</u>: As we described before, each public building has a list of residential building uses it. For each type of public building, we sorted the building in this category according to the importance of the building. The most important building is the one that is small in compare to the number of its users, so it is 'less satisfying' (being short building is second priority). It is calculated by:

$$overallArea(B) = (init\_height(B) + extra\_height(B)) * area(B)$$

$$\forall\, publicType: Sort\left(\left\{Building_{publicType}\right\}_i\right)$$

By calculating the building score for each $BP \in Building_{publicType}$

$$BuildingScore(BP) = \frac{overall\_area(BP)}{\sum_{B \in Buildings_{resd}(BP)} overall\_area(B) * init\_height(BP)}$$

$Buildings_{type}$ is the restriction of the full list of buildings data to those in the specific type(s). $publicType$ are all types but residential buildings ($resd$).
$Buildings_{resd}(BP)$ are all the residential buildings who linked to this public building, BP.

This procedure is done separately for each type: sorting buildings as described above, add floors to the most important building, recalculate this importance and sort again. We add floors to buildings only if we haven't reached the allowed max height, if we reached the maximum, we continue to the next public building in the importance list.

We used this procedure of simplification (instead of doing the same as we did for the residential), because we wanted to connect public building to residential building and because we wanted to reduce the heavy calculation which are more critical for the residential buildings.

3.3. <u>Score Evaluation Function</u>: Both algorithms find a solution that maximizing the score, while the objective is a weighted function comprised of:

a. **Needs:** the weighted average of satisfaction with the required needs. It is not always beneficial to add floors to public buildings (due to height limitations or floor area being too large in comparison to the needs).

$$needs(type) = \frac{\sum_{B \in Buildings_{type}} area(B) * extraHeight(B)}{metter2units_{type}}$$

$$\boldsymbol{PlanNeedsScore} = \prod_{\substack{type \in \\ buildingTypes}} \min\left(\frac{needs(type)}{extra(type)}, \frac{extra(type)}{needs(type)}\right)$$

Where $needs(type)$ is the area needed to add for this type of building, for public it is what we calculated according to the 'guide for allocating public buildings'. Residential needs are determined according to the number of housing units doubled by the area of a residential-unit. $PlanNeedsScore$ takes the min among both options, so we don't take more either less than needed.

b. **Distances:** the weighted average of the satisfaction according to distances from the public services. Here we calculated over all public buildings and not restricted only to the linked as described before (before we used it to simplify the calculation of the public extension plan, but here we just want to evaluate a state of a city, so in this aspect, adding floors even to farther building is more helpful than doing nothing)

$$SumWeightedDist(PB) = \sum_{\substack{B \in \\ Buildings_{resd}}} \|location(B), BP\|_2 * overallArea(B)$$

$$AvarageDist(publicType, PB) = \frac{SumDist(PB)}{\sum_{B \in publicType} SumDist(B)}$$

$$\boldsymbol{PlanDistanceScore} = \prod_{\substack{type \in \\ publicTypes}} \left(\frac{\sum_{\substack{PB \in \\ type}} AvarageDist(type, PB) * \left(extraHeight(PB) + area(PB)\right)}{\sum_{\substack{B \in \\ type}} \left(extraHeight(B) + area(B)\right)} \right)$$

$SumWeightedDist$ is the weighted (according to the area) distance of all residential buildings ($Buildings_{resd}$) from a specific public building. $AvarageDist$ is the ratio between a specific public building's $SumWeightedDist$ value to the sum of all public buildings' $SumWeightedDist$ value. $PlanDistanceScore$ is the score of all the plan according to the weighted distances.

c. **Cost:** budget limitations of the plan; Here, buildings over nf_ext = 12 floors or building with extension of more than p_ext = 20% of the existing height is more expensive.

$$\text{cost(B)} = \begin{cases} if\ extraHeight(BP) > \alpha & \beta \\ else: & 1 - \max\left(extraHeight(B) - \lceil \gamma * initHeight(B) \rceil, 0\right) \end{cases}$$

$$\boldsymbol{PlanCostScore} = \prod_{\substack{type \in \\ buildingTypes}} \prod_{\substack{B \in \\ Buildings_{type}}} (1-\varepsilon)^{\text{cost(B)}}$$

Where $\alpha = 12$ is the number of floors where the cost is dramatically increased, then the fraction score decreases in the power of $\beta = 0.9$. Increasing a building by more than (100$*$ γ)% of the building height (original) will increase the cost. In this case $\gamma = 0.2$, and $(1-\varepsilon) = 0.95$

The final Score was a weighted function between these evaluation functions. After checking each one separately and checked the results, we saw better solution for Needs, close to this was Distance, and last the cost.
We fixed *planNeedsScore=0.5, planDistanceScor =0.4* and *planCostScore=0.1.*

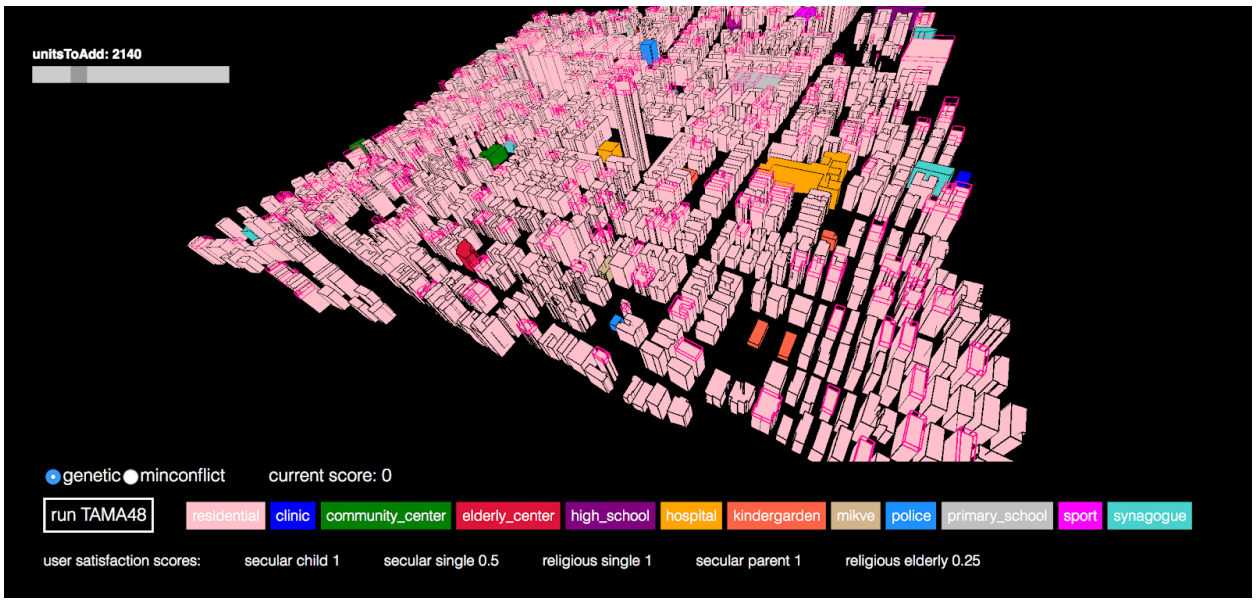
4. Graphic User Interface

In our GUI we show the user a 3D graphic representation on the algorithms' output, displaying the added floors to each building in our map, along with the score, each one of the algorithms outputted. After the user sets the amount of units he wishes to add and selects the algorithm to run, the added floors that are calculated are drawn in a different manner from the original buildings in order to ease the visual conception.

The GUI is written in JavaScript using the p5.js library as the main visualization tool, WebGL as the 3D rendering engine and easycam.js library as the in-app spatial navigation system. In order to connect the JS with the Python algorithms, since we used multiple external libraries in Python we couldn't use various methods of browser python implementation. Instead we built a Python server using Flask microframework to communicate between our GUI and our main().

Each building is an array of sections, each section has a polygon and height, either initial or added. The buildings are parsed from a json file that is updated after each code run with the new added floors. This method was chosen over importing the .obj 3D file of the buildings to downsize the overall program size and runtime. For each building we build a shape from its ground shape vertices, due to the fact that each building has a unique shape and isn't simply a box. This too lowered the amount of data we need to hold on each building since the vertices hold the absolute location of the building, instead or the relative location, thus negating the need for area and location data.

Although initially intended to be more, we decided to minimize the parameters given to the user to control, to the unit counter and the algorithm selection, in order to ease the use and let the user focus on the output rather than having “too-much-to-decide”. The remaining parameters were hard coded in a manner that they can easily be displayed and used in the GUI.



How are you going to test your results?

In addition to the score evaluation function and what analysis of the results as described before, we also used a different method of evaluation, which we call the personal satisfaction evaluation method. We specify five types of people, and for each type of person, we create an algorithm that calculates his satisfaction as a function of the public buildings he uses. For each person, we randomly select a building for its residence, and calculate its satisfaction, according to that person’s needs. Each type of person’s satisfaction is dependent upon its degree of proximity to different types of public buildings. We analyzed the needs of different types of populations, whether a child, a parent, single, or elderly, and whether or not that person is religious, and we found out the preferred distance for that person to the public buildings relevant to that person’s station in life. This evaluation method allowed us a micro-scale view of our results.

Why do you think that your approach is the right now?

As described above about the algorithms and in compare to what we described by details in the *First Report*- about CSP algorithm. We found out that CSP algorithm is not good for our case: the representation of the states (goal and init) should be defined in advance, and the number of states is num-of-optional-floors in the power of number-of-buildings, exponentially in a very large number. The solution we presented of iterations for finer resolution is semi-optimal, not flexible.

Moreover, the constraints are rigid (while in our case we just want to give lower score rather than omitting the options), and not necessarily local constraints, in that mean, not all action can be defined by pre-add-delete, and the transformation between states is not clear for reverse engineering.

The reason we chose these two algorithm for solving is that they allow us more flexibility and as described it is more natural for our data and problem. The method of searching a solution is very different between the algorithms, so it is more interesting to compare between them (with the same evaluation function). We also guessed that it is make sense to combine between good results from the previous iterations. We thought it can help us to narrow the search space and at least fastly find a nice solution. Since this data is such a mess, and after evaluating our solution in several ways, we think that this guess was good, while the other solution of min conflict requires more research of how to define a conflict. But its idea of gradually adding floors is nice, convenient to track and might be good.

Thank you!