Assignment 3

COMP4418 - Vincent Rubingh

Decision variable(s)

The decision variable used is an array of a set of integers (1..nTables), *TableAt[i]* with index from 1 to *nGuests*, that keeps track of where every guest *i* is seated (at what table). For example guest 3 is seated at *TableAt[3]*. Every guest in the array, can take up the value between 1 and *nTables*.
First I tried creating an array consisting of sets similar to *Pref* and *Groups*. Unfortunately, I could not get it to work like that. After checking out some examples, I adopted one array to keep track of guest seating instead.
I think this proved to be an advantage with the constraints, because it made formulating them quite easy and succinct. As far as adding more decision variables, that never seemed like much of an option, since one array does the job nicely. Additionally, having the domain for the array be 1..nTables made the problem space considerably smaller than using 1..nGuests for the domain of another array. The only problem was that creating the output was difficult, because it was impossible to know when the last person at a table would be through the iteration, so I had to create an extra variable, *output_table*, to handle this.

Constraints

Max and min per table

The first constraints put both a minimum and maximum on the amount of people at each table. I wasn't sure how to encode this but luckily the Scene Allocation example from the lectures had a similar constraint. So by using *atmost(nSeats, AtTable, table)*, we ensure that there at most *nSeats* occurrences of each table integer in the *AtTable[]* array.
Similarly by using *atleast(floor(nSeats/2), AtTable, table)* we ensure that every table has at least $\lfloor nSeats/2 \rfloor$ guests.

Everyone seated

I realized after trying these constraints with maximizing utility, that there was no constraint that ensures every passenger is seated. To achieve this, the model iterates over all the guests in the AtTable[] array and makes sure that all values are assigned a value.

Grouping

The grouping constraint ensures that guests in the same group will be seated together. This is achieved by first iterating through the different groups in *Groups,* and then iterating through the individual passengers in those groups. Every group member is compared to the first person in the group to assert they are at the same table.

Trouble

Similarly, to grouping, the troublemakers are constrained by being compared against each other to prevent them from sitting at the same table. It iterates through all the troublemakers with two different iterators, and compares the guests against one another to make sure they're not at the same table.

Testing

As for the testing procedure of the solve time, I tried to run the test 50 times for each number of guests, and both the evaluation functions, and calculated an average. The only exception to this is the utilitarian function with 30 guests, which took over 2 minutes to calculate.

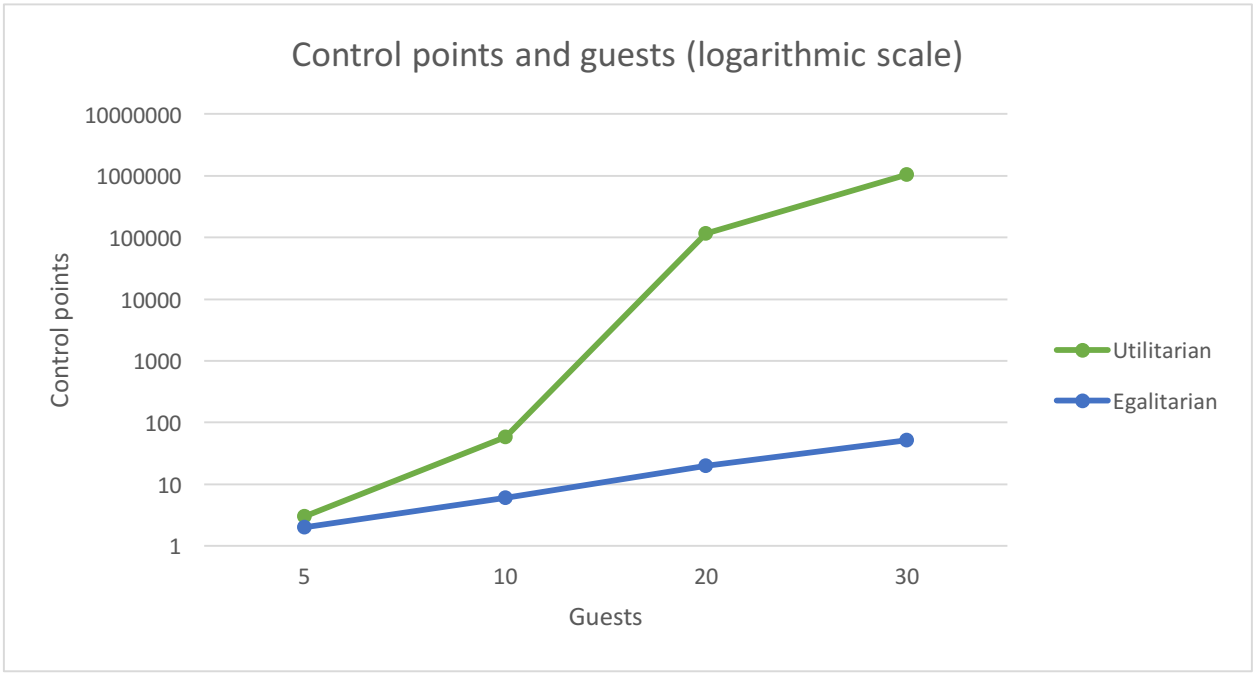
Solve time and control points for evaluation functions

# guests	5	10	20	30
Utilitarian solve time (in msec)	49.25	54.2	74.45	123000
Egalitarian solve time (in msec)	48.15	52.25	55.75	58.3

# guests	5	10	20	30
Utilitarian - control points explored	3	59	115986	1055625
Egalitarian - control points explored	2	6	20	52

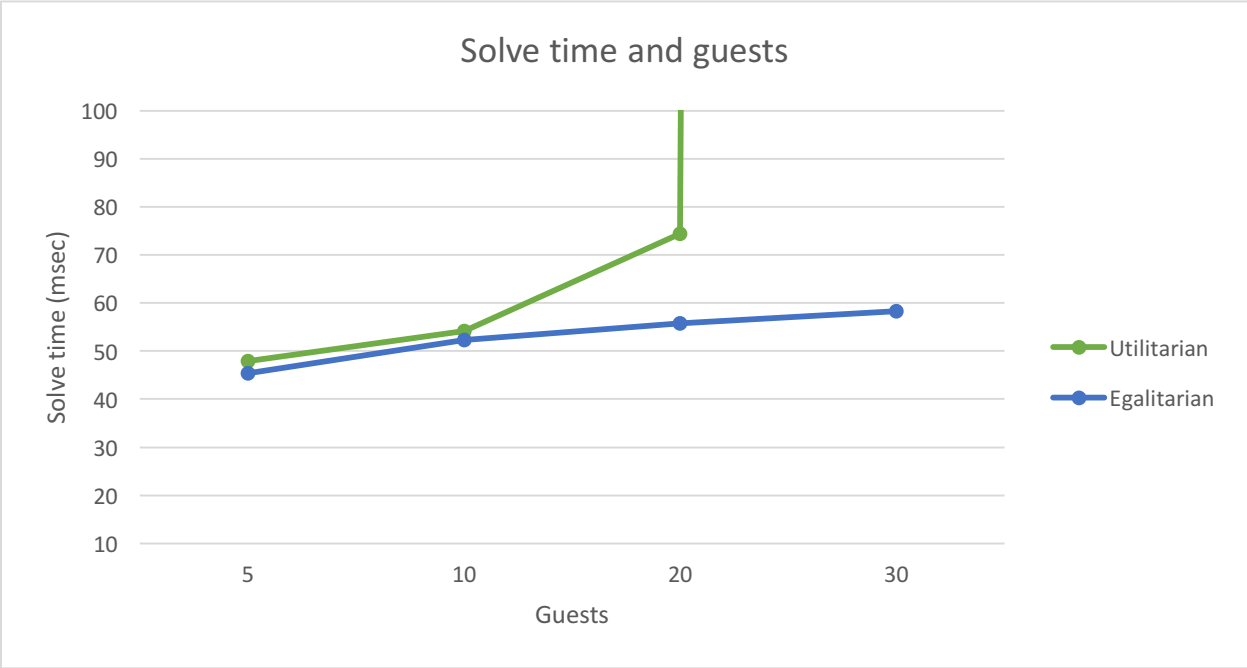
The utilitarian evaluation clearly takes a lot more time to solve than the egalitarian model, as evidenced by the higher control points throughout, and higher solve time as the number of guests increases, compared to the egalitarian evaluation. My guess is that this is due to the egalitarian function having to find the minimum of the sum of preferences for a guest, so every guest preference with a higher sum can be pruned before being explored fully.

Conversely, with the utilitarian calculation, it needs to take into account the sum of the preferences of all guests (and calculate them all), which clearly requires a lot more calculation and results in a lot more control points to explore.



Increasing the amount of groups had a negative effect on the solve time for the utilitarian function. At first I did not expect this, but then I realised this is due to less constraints, which translates to more possibilities to explore, since they can't be pruned now. The difference can be quite large, when adding a few groups of 3 for the 30 guest example, the solve time could be brought down from over 2 minutes (my initial configuration) to less than a second. Similar to the groups constraint, increasing the number of troublemakers decreases possibilities, but it was a fairly minor effect compared to the groups.

Since the solve time for utilitarian function was such an outlier I considered changing the complexity to fit more with my expectations, but then I would be choosing the problem to fit the data I expect.



note: showing the 30 guest point for utilitarian was not a solution because nothing else would be visible, so instead I left the data point out of the graph

One note I have to make with regard to the tests, is that this data is somewhat limited. For example, with 10 guests there could be 3 tables of 4, and less groups than there will be with 20 guests and 5 guests at 4 tables, which all makes it hard to compare. The data I chose determines the difficulty of the test to a large extent. In short, with different data for different guests, it makes it harder to compare, although it does provide a good indication. Lastly the solve time calculations have a big margin and deviation, especially when they are in the lower range and are susceptible to other processor activity.