Wedding Seat Problem

Author : Victor Choudhary

Problem

The problem has been described in the Assignment pdf provided by the university. In abstract form it can be described as follows :-

> "*nGuests* are invited in a party, with *nTables* available each having capacity of *nSeats*. We have to provide a seating arrangement as to maximise either *Utilitarian happiness* quotient or *Egalitarian Social Welfare happiness* quotient."
>
> Some of the major constraints are as follows :-
>
> - no table has more people than nSeats.
> - every table must have at least floor(nSeats / 2) people at it.
> - every person must be seated in the groups.
> - everyone in the trouble set must be seated at different tables.

Modelling the Problem

Decision Variables :-

- array[Table] of **var** set of 1..nGuests : **t**
  - This decision variable is a mapping of table to guests sitting on that table. It is basically an array of set of guests sitting on that table. As each table will carry unique number of guests, it is best to represent them as a set of guest, it will help both conceptually and computationally, as we can use the underlying functionality of sets and their operations.

- **var** int: utilitarian
  - This decision variable is the summation of utilitarian happiness calculated for each of the guest. It is used by the solve maximise tool in order to maximise this variable.

- **var** int: egalitarian
  - This decision variable is the summation of egalitarian happiness calculated for each of the guest. It is used by the solve maximise tool in order to maximise this variable.

Constraints :-

- **constraint forall(i in Table)(card(t[i]) <= nSeats /\ card(t[i]) >= floor(nSeats div 2));**
  - This constraint is used to make sure that each table is neither less than half populated nor overpopulated.
  - The constraint goes to each table and checks the cardinality of the table.
  - In real world, it corresponds to comfortable arrangement of guests per table.

- **constraint forall(i in 1..nGuests)(exists(j in Table)(i in t[j]));**
  - This constraint is used to make sure that each guest has been seated.
  - The constraints simply visits each guest, and then in inner loop for each guest visits every table and make sure that there exist at least 1 table where the guest is seated.

- **constraint forall(i in 1..nGroups)(exists(j in Table)(Groups[i] intersect t[j] = Groups[i]));**
  - This constraint makes sure that groups are seated together.
  - The constraint visits each group provided as input, then as in inner loop visits each table and does the intersection with the guests sitting on table and the corresponding group.

- **constraint forall(i in Table)(t[i] intersect Trouble != Trouble);**
  - This constraint is used to make sure that trouble makers are not seated together.
  - The constraint visits each table and makes sure that the intersection of guests sitting on that table and the troublemaker set doesn't match.

- **constraint all_disjoint([t[i] | i in Table]);**
  - This constraint is used to make sure that there is no guests sitting at more than one table.
  - It simply makes sure that all the tables have disjoint set of guests sitting at them.

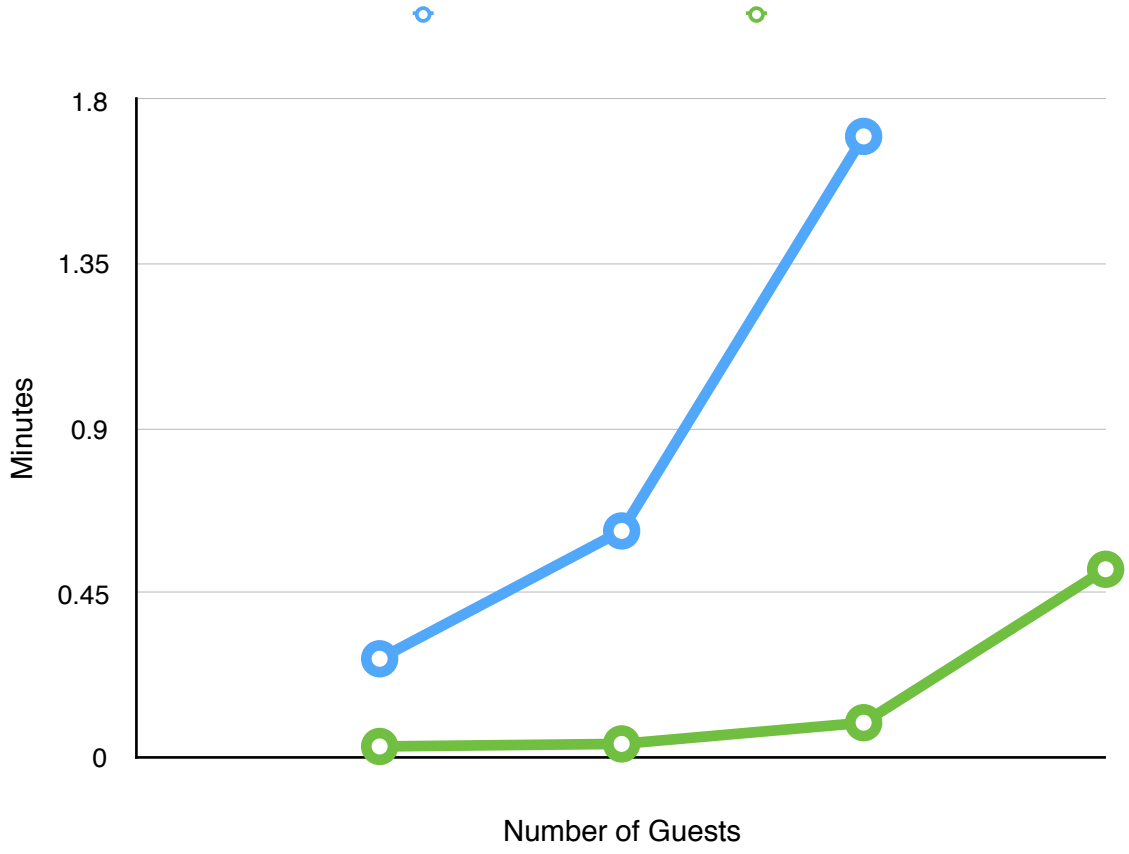
- **utilitarian = sum(g in 1..nGuests) (sum(i in Table) (if g in t[i] then card(t[i] intersect Pref[g])*PrefWeight else 0 endif));**
  - This constraint is used to measure the utilitarian constraint.

- **egalitarian = min(g in 1..nGuests)(sum(i in Table) (if g in t[i] then card(t[i] intersect Pref[g])*PrefWeight else 0 endif));**
  - This constraint is used to measure the egalitarian constraint.

Testing

Blue Line - Utilitarian Happiness
Green Line - Egalitarian Happiness

By increasing the number of guests, the summation constraints of utilitarian happiness which are going through each guests gets a increase in the branching factor for the search, thus increasing the overall computation time.
The time taken for computing utilitarian happiness can be seen significantly higher than



egalitarian happiness.