Vaughn Bangash, Sam Traylor

Intro to AI

Project 4: Constraint Programming

4/13/21

Question 1&2: The decision variables in our model are the tables within the SolvedSeating array. These tables are stored as sets, where each integer of the set is a seat, its value corresponding to which guest is there. It was an interesting challenge to sort out how the guests are able to sit rather than rearranging the order in which they are already seated. We first attempted this with an array of arrays each representing a seat but quickly found that the language and operations were better suited to an array of sets. Using the tables as decision variables allows us to address table-wide constraints much more easily than a 2-d array, and more easily than a 1-d set with no indexing. We can easily say “is someone from the set trouble in this set” with the ability to address a table directly, and without having to mess with indexing. To tackle the problem of preferences we had to reevaluate our decision variables and create a new one representing a reverse model, the inverse of our other decision variable. Initializing this variable then channeling the two together using the int_set_channel operation allowed us to use the reverse model in our preference function. We used this array of guests mapped to tables to find the intersection between a given guest’s table and their preferences.

Question 3: Our first constraint was used to link the reverse model ReverseSeating and our array of tables SolvedSeating. Using int_to_channel we ensured that the arrays were counterparts of each other, with SolvedSeating indexing people by table, and ReverseSeating indexing tables by

people. Our next two constraints ensure that the tables have the proper number of seats by stating that the cardinality of the table set is less than or equal to nSeats and greater than or equal to floor(nSeats/2). This was our first plan to implement it and we didn't see any flaws. Our next constraint ensures that all guests 1..n are present and accounted for. This was simple and achieved by requiring that all values in our array of 1..nGuests have a spot in the union of all the sets in the SolvedSeating array. The constraint after this uses the all_different operator to ensure that all guests seated at our tables (sets within SolvedSeating) contain no identical guests. This also functioned as a slight form of value symmetry breaking, ensuring that all guests have differing values.

We first tried to implement the constraint for groups being sat together by using nested forall loops to iterate through tables and groups ensuring each group was either a set in the array of tables, or a subset of one of the existing sets. The tables being decision variables made it difficult to easily constrain the group to be in the set using built-in operations such as 'in'. To get around this we included a line "disjoint(g,t) xor ('subset'(g,t))" which checks that the group and table either have no intersection, or g is a subset of t. Our next constraint was used to ensure that no two troublemakers are seated together. The constraint iterates through tables, and the nested loop iterates through Trouble, constraining each two non-same elements of trouble such that its impossible they're simultaneously at a table. The non-same specification was needed because otherwise the loop starts by comparing variables i,j holding the same value, and when you say they can't be in the same table that becomes unsatisfiable.

We first tried to implement the preference constraint using single double and triple nested forall loops trying to ensure that for the guest at the table if there was an intersection between the guest's preferences and the table the guest was seated at, the cardinality of that intersection

subset would be multiplied by the preference weight and added to Utilitarian welfare. Our current implementation still iterates through the guests and tables but using the built-in sum operation rather than the forall loops. To find the happiness values of each guest, we take the cardinality of the intersection of both the preference set for our guest, and the table they're seated at. We again multiply that number by the preference weight to be totalled up for the utilitarian value and find the minimum to get egalitarian. We now have these set up under two different constraints, one for Egalitarian and one for Utilitarian. The only difference being that we find the minimum for the former and take the sum for the latter. Lastly we implemented a constraint to further ensure value symmetry breaking by using the increasing operator to impose ordering. This ensures that no separate solutions can be comprised of the same guests at each table, just in a different order.