Timetable solver - ttsolve
Used algorithm

Paweł Parys (pp209216@students.mimuw.edu.pl)

January 24, 2004

1 General idea

In my program there are three basic treatments:

- Constraint propagation. It is some kind of deduction. I try to determine what assigments of activities are possible or impossible.
- Searching (backtracking). I try to use all possible timeslots and resources.
- Greedy improoving. When I have a solution I move single actvitities to get better score.

First two actions are done simultanously, after every recurencion step of backtracking I'm doing constraint propagation, so that the new assumptions are leading to new conclusions. When a solution is found I use the third action to improove it a little bit.

2 Whole program

The program repeats a few times single steps, which are almost independent and identical (I call them ,,approaches"). In every approach I build new solution, but basing on the previous one. I always remember the best achieved solution — when I come to worse solution it is omited. Generally I have three stages:

1. In first approach I build any possible solution from scratch. I accept also some incorrect solution (with some unassigned activities), but with very big score, so that they are unprefered. In that way we always have a solution.

2. Then I allocate anassigned activities (if any). I take one such activities and activities dependent on it (using the same resources) and I try to change their possitions, so that the one additional activity can also be allocated (I leave the other activities as they are). It is repeated until all activities are allocated or until nothing changes.

3. Then I do the same, but with random activity (and activities dependent on it), so that better timetable can be constructed. This is also repeated several times.

3 Single approach

3.1 Simplified algorithm

At the begining I will concentrate on arranging the activities, so that there are no collisions. Choosing resources from groups is only a addiitonal feature and will be described later.

On the whole every approach works as described in introduction. It is reccurentional procedure. In every step of reccursion I decide about one start time of activity. For each activity and for each time I remember if this activity can start at this time or not. This state I call ,,space". Sometimes the start time of activity I call ,,variable". The space has to change while going into reccursion. At each step of reccursion I copy the current space and I operate on the new one. (Other possibility is to remember changes and undo them when going back.)

I can remove the start times which are forbiden because of time preferences. I do this at the very beginning. The other constraints are solved using propagators. A propagator is a special object that ensures given relation between two variables. A propagator binds two variables. When the set of possible values of one of them becomes limited (especcialy when it becomes only one value), then we can eliminate some values of the second one. When two activities uses the same resource, then they can not overlap - I create one propagator for each such pair. There can appear also some dependences (eg. that a activity should be arranged before other one), they also need apropiate propagators.

Whenever a set of possible values of variable changes (decreases), then all propagators concernig this variable should be racalculated. But I don't do that immediately, but only when needed. Thanks to this lazy algorhith, when there are more changes of the same variable, propagars are calculated only once. When the propagator was calculated after the last change of its variables, we say that it is ,,stable". Then all propagators are stable, then we can not change anything using the propagators, and we say that the space is in ,,stable" state.

The propagation strategy is done as follows: With every variable I remember the last time when it was modified and with every variable I remember the last time when it was actual. I also have a queue of unstable propagators and a queue of modified variables. Every unstable propagator is in the queue or one of its variables is in the queue. When we change a variable, we update the modified time and if we put it into the queue (if it is not there). When we need the space to come into the stable state, we have to recalculate all unstable propagators. Having these two queues we can easly find unstable propagators. We take all propagators from unstable propagators queue. When this queue is empty we take a variable from the second one and we put propagators binding it (but only these realy unstable) into the first one. Of course during the calculation of propagators, some other may become unstable, so we repeat the operation until there are no changed variables in the queue.

The single propagator do only very simple oprations. It uses only the minimal and maximal values of variables (but affects on the whole set of possible values). It has significant effect only when the segment of possible values of one of the variables is already small. A propagator can also remove itself, when it see that the condision it checks will be always true (for all values that are still possible). When a set of possible values of a variable becomes empty, then the branch of searching fails, we have to go back in the reccursion.

Choosing resources from groups is done as follows: After a start time of a activity is set to one specified value, we select which resource should be used by this activity. We have a array which for every resource (not every, only for these, which appears in groups) and every time slot remembers if it is used or not. We also have a array which for every group and every timeslots remembers how many resources are still unused. Using these arrays we can easly find which resource from group should be used. When trere are no resources left in group at specified time we can disallow other activities needing the group using this timeslot. In fact it is more complicated, because the same resource can appear in other groups or can be used without any group.

4 Future work

1. It should be good to add some global propagators. They will bind all activities using specified resource. They will be checking the quantity of timeslots needed and available. When some activities can be whenever and some can be only in short period, then the first ones can not be at this period.

2. Another seperate heuristic of rearanging activitities: Something similar to maximal matching algorhitm. I will try to move the activity

to better possition, and if there is collision with another one, then the second one will be pushed somewhere else and so on. I will remember all the reached states (using the hashtable), only states quite similar to the initial one are allowed.