README

HOMEWORK 1 AI

ILINCA SEBASTIAN-IONUT 331CA

Constraints and states representation

Functions already given in utils are used to represent constraints; although those may seem sufficient, the result from parsing the input files was modified, adding intervals that are more precise and can be used directly for computing cost(e.g. if a teacher has ’10—14’ the added values to the dictionary that contains constraints and additional information necessary to create the timetable will be ’10-12’ and ’12-14’; this also applies for information like ‘!10-14’). With this implementation, we sacrifice some memory in order to compute cost faster. The representation of constraints are englobed within the dictionary already given by parsing the yaml file and will be used in the ‘cost_solution’ function from ‘helper’ module. As there are also hard constraints, these will be checked from the beginning of generating timetables, using also the dictionary, a classroom needs to be dedicated for the subject, teacher need to be specialized on that subject etc., but there are also hard constraints that are implemented in the code with ‘if’ statements: such as, the teacher cannot be in the same interval twice, less than 7 lectures for each teacher, etc.

As for states representation, I choose my own form of dictionary, making it later on the exact form needed by ‘utils’ module. My timetable follow the rule: make a dictionary that have for each day given a dictionary, that dictionary has for each interval another dictionary, and that dictionary has for each classroom a tuple that contains two pieces of information: subject taught and teacher, in case for an empty classroom, tuple (0, 0) is used.

Optimizations made for Hill Climbing and A* algorithm

Random Restart Hill Climbing

This algorithm is preferred as an alternative for the classic one. An optimization made for this was not generating all the neighbors possible, instead, generating all the tuples(that can be switched), shuffling them and then iterate from all, we can now save memory but also time, regarding the fact that after building a state from a tuple and observing that this new state is at least with 2/4(can be your choice here) points better. We do not need to made the other states from those remained tuples because we are satisfied with this next state and also will save us time.

Another optimization that is worth telling about is the creating method of initial states. Being present three methods: ‘get_random_state_based_teacher’, ‘get_random_state_based_teacher_break’ and ‘get_random_state_based_intervals’, we can

create a random choice between each for each restart(the second is being used only when teachers prefer breaks). Function ‘get_random_state_based_intervals’ will create when assigning students a probability greater for classroom available with a bigger capacity, then choose an interval from the pool created. On the other hand, the remaining two functions are considering the teachers’ options very valuable and for each subject they’ll create a list of teachers and intervals available for each one, but depending on how good that interval is(less constraints not respected is always better), the higher the probability to be picked. For the bonus, the timetable made until that moment(the work is in progress, random state not finished) will also review it and add a constraint not respected(so a less probability to be chosen) for that specific interval if break option does not appear to be all right in the ‘next possible interval’.

A*

One of the most important optimization about the implemented algorithm is that it gives up on exploitation of a state if it is not worth it. This means that if I take out from list a state and realize I cannot fit all students remained in this timetable even if the classrooms would sustain every subject or there is a subject that can not be fitted for everyone in the remained specialized classes.
Another optimization is that not all next states are put in the priority queue, the maximum is 10(can be changed); also, if there is at least one next state that does respect until that very moment all the soft constraints, those states that does not follow the soft constraints will not be added to the queue. This method is a good representation of: ‘If you are going to be wrong either way, be wrong later’.
The modified algorithm is sort of optimistic, because, in case the iteration limit is exceeded, it tries to solve the remained timetable regardless of how many soft constraints are breached; but after some tries, if it cannot respect all the hard constraints, it will back up and return the best state where not all students have been allocated when the iteration exceeded(before calling the function ‘try_make_it_work’).

Comparison between the two algorithms mentioned above

LEGEND:

T -> Time
S -> Generated states
Q -> Quality of solution(less is better)
H -> Hard constraint

Test	Hill Climbing	A*
dummy	T: 0.002166; S: 1; Q: 0	T: 0.0543487; S: 118; Q: 0
orar_mic_exact	T: 0.695995; S: 2952; Q: 0	T: 2.739864; S: 2537 Q: 0
orar_mediu_relaxat	T: 0.726713; S: 3; Q: 0	T: 28.9338181; S: 18559: Q: 0
orar_mare_relaxat	T: 0.96228; S: 1521; Q: 0	T: 66.1880571; S: 20838; Q: 0
orar_constrans_incalcat	T: 316.1237008; S: 1154874; Q: 0	T: 691.2545630; S: 681380; Q: 1(H)
orar_bonus_exact	4, 0071218; S: 5; Q: 0	T: 548.5660459; S: 298879; Q: 1(H)

The results from A* where a hard constraint is breached(‘orar_constrans_incalcat’ with SO not covered and ‘orar_bonus_exact’ with MS not covered) happens due to the need of a ‘near perfect’ matched solution, and if the A* choose not to follow the correct path, but deviates a little

with another intermediate, it will try hardly to remediate the mistakes produced, not knowing though where the things are destined to fail, and can be far away.

An optimization for better results as quality of solution can be made by using closed list to ensure that no duplicates are evaluated and exploited. It's commented in the code those lines that do that, but unfortunately I did not have the time needed to run those tests once again(very little time till deadline), but I am sure that these lines of code would made a huge difference and the results would be following the hard constraints.

Conclusions

From test results, we can see that the problem can be interpreted easier with Hill Climbing algorithm(at least Random Restart version, the classic one surely will get results that breach a lot of soft constraints), being also faster than A*. Of course, if the heuristic would be perfect, there can be a little bit of sense in telling pros and cons for those two(e.g. the user can be unlucky and there will be a lot of random restarts, although it will be very uncommon due to the optimizations specified just earlier).

As final statement, we observe that the algorithm used to search in the space of states depends a lot on the type of the problem, as here is better to use Hill Climbing, for a maze instead A* surely dominates.