Homework 6

Course Scheduling

The university offers courses each term. The challenge is to assign courses to time slots so to satisfy the numerous constraints. This problem is currently solved by humans, but it is very difficult and often leads to scheduling mistakes. Such a program could eventually assign course meetings to available classrooms, but that is a future extension. (Note it is always good to think about future extensions, but to be explicit that we are not going to do them now.)

Scheduling courses to timeslots is a constraint satisfaction problem that is NP complete. There may be multiple possible solutions, but the problem space is so large, finding even one solution can take more time than we have. We can use an effective AI algorithm to find one of the many solutions.

We can simplify the problem while still planning for the future extensions. For example, we can ignore room assignments and we can schedule courses for one specific school or department. We can further divide various constraints into phases to avoid being overwhelmed by the complexity of the problem.

Source of Constraints

- Specific instructors are assigned to specific courses (their lectures can't be at the same time).
- Lectures may have labs and/or discussion sections (these can can not be at the same time as lecture).
- The university publishes suggested course plans for students by major to help them choose which courses to take each term (so these courses should not conflict with each other).
- Additionally, instructors have preferences for when they would like to (or when they are available to) teach.

Input Files

inputSchedule is a list of the courses, their initial day format (e.g. MWF, TR) and time slot (9:10am, 24-hour format) plus duration in minutes. The initial file could be the schedule from the previous year, but with new courses added. Time does not matter, e.g., all the courses could be scheduled at the same time (MWF 8am). Other courses should be moved to minimize the number of constraint violations. Courses are scheduled in 30-minute slots with a ten-minute break before the start of every course. (So a course can be 50, 80, 110,… minutes long and start at 9:10, 9:40, 10:10, 10:40 and so on).

```
course => I&C SCI 31
type => LEC
days => TR
start-time => 910
duration => 80
instructor => KLEFSTAD

course => I&C SCI 31
type => DIS
days => W
start-time => 910
```

```
duration => 80
instructor => KLEFSTAD
```

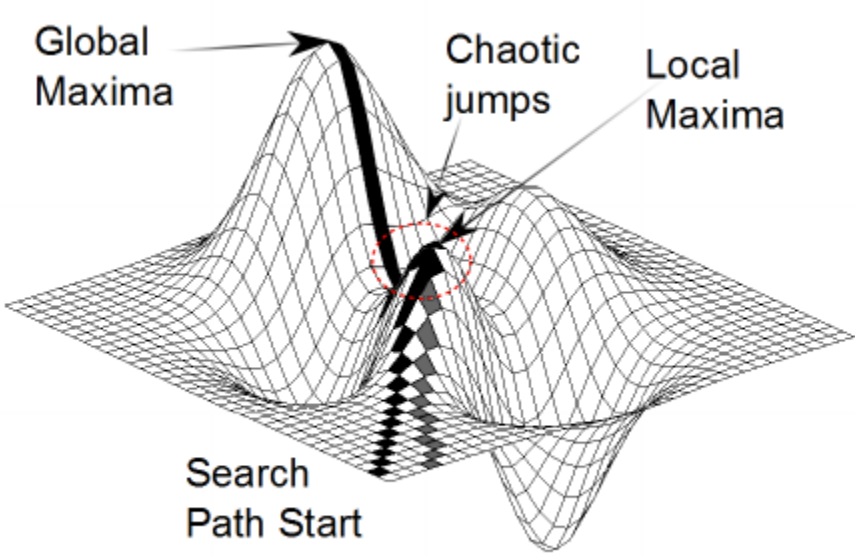
coursePlans lists courses that should be taken together.

instructors lists the teaching preferences/availability of each instructor, e.g., Klefstad likes to teach between 9am and 3pm.

Phases of Program Design/Implementation/Testing

Phase 0: Read inputSchedule.txt, coursePlan.txt and instructors.txt into a data structure, then print the inputSchedule back out in the same format to standard out.
Phase 1: inputSchedule is processed so that lectures, labs, discussion do not conflict with each other. Note outputSchedule is in the same format and may be used as input in the future.
Phase 2: coursePlan is considered so that courses suggested together do not conflict with each other.
Phase 3: instructor preference is considered so that an instructors courses are within their preferences

Approach: Hill Climbing



A given problem may have many solutions. However, the number of constraints is quite large and difficult for a human to solve. It is an ideal application of Hill Climbing which is based on Gradient Descent (or Ascent in this case). An optimal solution lies at the peak of some mountain in the problem search space, e.g., the Global Maxima in the figure above. Start with some solution (even a really poor one) and evaluate the quality of that solution using an evaluation function. To find the peak, look at all (or each one in turn) of the neighboring points. If a change improves the current solution (move upward) make that move. The algorithm can pick the best improvement (expensive, Discrete Space Hill Climbing Link) or just any improvement (easy, Greedy Hill Climbing Link). One weakness of Hill Climbing is that some problem spaces have local maxima (smaller peaks) where we can end up and get stuck. If we get on to one of these local maxima humps, we may never find the global maxima because that would require going down hill for quite a while to cross the saddle point. There is an approach to dealing with this problem called Simulated Anealing. We can safely ignore this for

now. The next step is to define the evaluation function, EVAL(State s). Here is a start on the algorithm (I will revise more before you code it)

```
void Discrete_Space_Greedy_Hill_Climbing(Schedule startState)
{
        currentState = startState
        currentEval = EVAL(currentState)
        for ( ever ) {
                changes = 0
                for ( possibleNext in NEIGHBORS(currentNode) )
                {
                        possibleNextEval = EVAL(possibleNext)
                        if ( possibleNextEval > currentEval )
                        {
                                currentState = possibleNext
                                currentEval = possibleNextEval
                                ++ changes
                        }
                }
                if ( changes == 0 ) break;  // go until we reach a maxima
        }
}
```

Interesting Design Issuses

We must choose data structures that are flexible yet allow us to solve each phase of the problem. We can (and should) use existing C++ STL containers/types, but must be careful to choose ones that will work for the full system (all phases).

The input files are in attribute/value pairs, instructor => klefstad. These files are generated from SpreadSheets. The information in the spreadsheet is important for the human dealing with course scheduling, but our programs are only interested in some of the information. We must keep the outputSchedule in the same format as the inputSchedule and we must retain the information and the order of this information. We can change the values of attributes, such as time and day (and maybe other information later as we extend our program).

Choice of Representation for the Schedule

We should identify likely operations on the schedule. Then we should identify the STL containers that will allow us to implement most of it quickly. We may need to add operations ourselves. Should we use inheritance or containment for our design units? It is a good idea to have a design unit for each problem unit - even if the design unit maps directly into a library unit. E.g., a Schedule may be implemented using a vector of lists, but we should still have a class Schedule. It may be publicly derived from vector<list<pair<string,string>>> to show implementation as a vector of lists of pairs of key/value pairs.

Meta note: This homework project is an experience building a larger scope program in incremental steps. We can share the design process, plan for testing, decomposition into functional subsets (phases), choices made for implementation. Note we deal with an open-ended specification which is normal for building real systems. I have never yet seen a real

software project where clear/complete requirements were written (or even known) before the system was designed and built. It isn't even possible to do so because of the complexity.

After you implement this program, can you calculate the time complexity of this scheduling algorithm? Let C be the number of courses to schedule (including all lectures, labs, discussions). Let L be the number of instructors. Let S be the number of time slots in the week. Let D be the distance in moves from where the start (where inputSchedule is) to where the solution (where outputSchedule is).