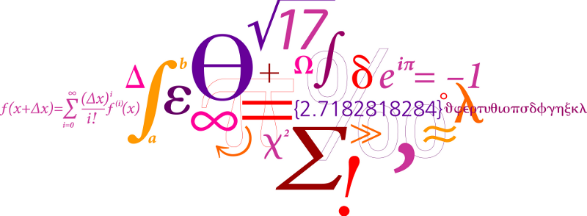


**42137 Optimization using Metaheuristics**

**University Timetabling**

Martin Wiboe

Burak Topal

Too Sheng Tack

**Table of Contents**

Abstract

1. Introduction
2. Problem Description
3. Implementation of Classes
   1. University TimeTabling
      1. Variables
      2. Functions
   2. Heuristic
      1. Variables
      2. Functions
   3. Classes Reading Input
      1. Basic Info
      2. Curriculum
      3. Courses
      4. Lecturers
      5. Rooms
      6. Schedule
      7. Unavailability
   4. XORShiftRandom
4. Metaheuristic
   1. HillClimber
   2. Simulated Annealing
   3. TABU
5. Implementation of Heuristic
6. Choosing a Suitable Neighbourhood Function
7. Delta Evaluation
8. Parameter Tuning
9. Appendices

**Abstract**

This report will investigate the implementation of Metaheuristic on the university timetabling problem and examine the outcomes. A brief introduction of the problem is initially outlined. Several Metaheuristic methods such as Hill Climber and Simulated Annealing with Taboo Search are applied to minimize and measure the objective value. Parameter tuning is also performed and the comparison of the results of each method is drawn.

**1.0 Introduction**

Every semester universities face the problem of creating good feasible timetable due to many complex constraints that have to be taken into consideration. Associated penalties will be assigned if the constraint is violated. The purpose of this report is to plan as many lectures as possible and avoid unwanted attributes to minimize the objective which is the penalty. Given a specific time limit, we can run Hill Climbing, Simulated Annealing and TABU to find the best solution (which may not be the best possible solution) to the problem. Providing the universities an optimization tool can help them to smooth out the planning and better utilize their resources.

**2.0 Problem Description**

This problem consists of weekly scheduling of the lectures of the courses within a certain number of rooms and time periods, where conflicts between courses are set according to the curricula published by the university. Due to the complexity, the formulation of the problem is simplified and more attributes could be added to enhance the quality of scheduling. In addition to that, this problem consists of several entities, constraints and components of the objective.

**2.1 Entities**

* Days, Periods and Timeslots

Given a fixed number of days per week and each day is split into a certain number of periods. A timeslot is a pair of day and periods.

* Courses and Lectures

Each course that taught by only a lecturer has a required number of lectures to be given and each course will be attended by an associated number of students. Lectures of each course must be spread into a minimum number of working days, moreover there are some periods that a lecture cannot be scheduled in.

* Rooms

Each room has a capacity, expressed in term of available seats.

* Curricula

A curriculum is a group of courses that any pair of the courses have students in common.

The solution of the problem is a number of assignments of lectures to a time slot (day and period) and a room.

**2.2 Constraints**

The following constraints have to be obeyed in order to produce a feasible timetable.

* **Lecture**

A predetermined amount of lectures is assigned to each course. Each lecture must be scheduled in different time slot and the total number of lectures cannot be exceeded.

* **Room Occupancy**

Each room can only accommodate one course in a specific time slot.

* **Conflicts**

Lectures of courses in the same curriculum or taught by the same lecturer must be assigned in distinct time slots.

* **Availabilities**

Some courses cannot be scheduled at specific time slots.

**2.3 Objective**

Several attributes contribute to the objective and each unwanted attribute has an associated penalty value.

**Unscheduled Lectures**

Each course has a lecture that is not scheduled will be penalized 10 points.

**Room Capacity**

For each lecture, the number of students attending the course must be less than or equal to the number of seats in the room. Each student over the capacity will be penalized 1 point.

**Minimum Working Days**

Lectures of each course must be spread into a minimum number of working days. Each day below the minimum working day will be penalized 5 points.

**Curriculum Compactness**

For any given curriculum, a lecture is considered as a secluded lecture if it is not adjacent to any other lecture from the same curriculum within the same day. Each secluded lecture will be penalized 2 points.

**Room Stability**

All lectures of a course should be given in the same room. Each extra room used will be penalized 1 point.

**2.4 Mathematical Model**

The problem is represented by a mathematical model. Sets and parameters are defined as well as variables, constraints and the objective.

**Sets**

C – The set of courses

L – The set of lecturers

R – The set of rooms

Q – The set of curricula

T – The set of time slots. I.e. all pairs of days and periods

D – The set of days

T (d) – The set of time slots that belongs to day d є D

C (q) – The set of courses that belongs to curriculum q є Q

**Parameters**

Lc – The predetermined amount of lecture of each course є C

Cr – The capacity of a room є R

Sc – Number of students attending course c

Mc – Minimum number of working days that the course should be spread to

Fc,t – 1 if the course c є C is available at time slot t є T, otherwise 0

X (c1,c2) – 1 if course c1 є C which is different from course c2 є C ( c1 ≠ c2 ) and conflicting, otherwise 0

V (t1,t2) – 1 if the timeslot t1 and t2 which belongs to the same day and adjacent to each other, otherwise 0

**Decision Variables**

Xc,t,r – 1 if class c є C is allocated to room r є R and timeslot r є R, otherwise 0

**Constraints and Functions**

Each course can at most be assigned one room at a specific timeslot and only if the course is available for the timeslot.

Each room can accommodate at most one course in a given timeslot.

Each course can at most be assigned to a maximum number of lectures.

Conflicting course are not allowed to be scheduled in the same timeslot.

The function Vt,r (x) indicates the amount of capacity that room r є R is exceeded in timeslot t є T.

Function Uc(x) indicates the amount of lectures by which course c є C is scheduled below the specified value Lc.

Function Pc(x) indicates the number of room changes by a course c є C.

Function Wc(x) indicates the number of days that the course is scheduled below the minimum working number of days.

Function Aq,t(x) determines if a curriculum in a timeslot has a secluded lecture.

**Objective**

The objective function is the summation of the penalties.

**Complete Model**

Subject to

**3.0 Implementation of Classes**

9 classes are created for easy maintenance which include University Timetabling, Heuristic, Basic Info, Courses, Curriculum, Lecturers, Rooms, Schedule, and Unavailability. Explanation of variables and functions created within the class will be discussed briefly as below.

**3.1 University Timetabling**

The purpose of this class is to read the data from given data files, find the best schedule which minimize the objective and write it into a CSV file for drawing chart and the purpose of parameter tuning.

**3.1.1 Variables**

Firstly, files such as basicFile, coursesFile, lecturersFile, roomsFile, curriculaFile, relationFile, unavailabilityFile are created to keep the information of input files. Then, a variable heuristic of type of Heuristic (self-defined class) is created to record the data from the files mentioned above. Followed by generating an initial schedule of type of Schedule (self-defined class) and searching for the optimum.

**3.1.2 Functions**

There are 2 functions operating in this class, which are the main function and startwithParameters function. In the main function, there are 2 options which the program can choose from to start running, one is solve the problem without benchmarking and another one is solve the problem several times and output the intermediate results. Meanwhile startWithParameters function operates as reading data from the input files.

**3.2 Heuristic**

This class is abstract and must be implemented according to the chosen heuristic. It will perform a search with the input schedule as a starting point and return the most optimal schedule.

**3.2.1 Variables**

Several variables are created such as basicInfo (type of Basic Info), curriculum (type of Curriculum), lecturers (type of Lecturers), courses (type of Courses), unavailability (type of Unavailability), rooms (type of Rooms) to contain the given data. In addition to that, other variables such as timeout (integer), countdownStartTime (long), iterationCount (int), int[] courseAssignmentCount are also created.

**3.2.2 Functions**

Numerous number of methods are created in Heuristic and some important functions are shown in the list below.

1. Public Boolean validateSameLecturerConstraint(Schedule schedule)

* Check that the courses taught by same lecturer are not scheduled in the same timeslot.
* Return true if the constraint is satisfied.

1. Public Boolean validateSameCurriculumConstraint(Schedule schedule)

* Check that the courses grouped in same curriculum are not scheduled in the same timeslot.
* Return true if the constraint is satisfied.

1. Public Boolean validateAvailabilityConstraint(Schedule schedule)

* Check that courses are not scheduled in unavailable timeslot.
* Return true if the constraint is satisfied.

1. Protected int[] getCourseAssignmentCount(Schedule schedule)

* Returns an array containing the number of times of each course has been scheduled

1. Public Boolean validateMaximumScheduleCountConstraint(Schedule schedule)

* Validate that no course has been scheduled more than minimum number of times
* Return true if the constraint is not violated

1. Public Schedule getRandomInitialSolution()

* Generate a random initial schedule

1. Public int evaluationFunction(Schedule schedule)

* Calculate the objective value of the schedule

**3.3 Classes Reading Input**

**3.3.1 Basic Info**

A function, Public void loadFromFile(String file) is created to read the number of courses, rooms, days, period per day, curricula, constraints and lecturers from the input file.

**3.3.2 Curriculum**

The purpose of this class is to keep the relation between the courses and curriculum. If a course belongs to a curriculum, the Boolean array isCourseInCurriculum[][] will return true. A public void loadFromFile(String curriculaFile, String relationFile, int numberOfCourses) function is created to read the data from curriculaFile and relationFile and store assignments as an array of Booleans.

**3.3.3 Courses**  
The purpose of this class is to store the information such as lecturer for each course, number of lectures of each course, minimum working days of each course and number of students of each course.

**3.3.4 Lecturers**  
Integer array, int[] lecturers is created to store the identity of lecturers. A public void loadFromFile(String file) function is created to read the data from a lecturers file.

**3.3.5 Rooms**

Integer array, int[] capacityForRoom is created to store the number of seats in a given room. A public void loadFromFile(String file, int numberOfRooms) function is created to read the data from a Rooms file.

**3.3.6 Schedule**

The purpose of this class is to store the schedule that expressed in term of [day][period][room]. A public void Schedule(int days, int periods, int rooms) function is created and first initialized all assignments to -1, which means that no course is assigned to the timeslot and room. Another method which is public String toString() will print the solution.

**3.3.7 Unavailability**

Boolean array is created such as boolean[][][] courseUnavailable which expressed in term of [day][period][course]. Return true if a course is unavailable in the specific timeslot. A public void loadFromFile(String file, int numberOfDays, int numberOfPeriods, int numberOfCourses) function is created to read the data from a unavailability file.

**3.4 XORShiftRandom**

It is a faster, higher quality replacement for java.util.Random. This class is called when generating initial random solution.

**4.0 Metaheuristics**

**4.1 Hill Climber Heuristic:**

Hill Climbing heuristic is an iterative local search method. It starts with and arbitrary solution and it tries to find a better solution by searching in the neighborhood. If it finds a better solution then it saves it as a current state then in the next iteration it start from there. The algorithm terminates If the time limit reached and return the best solution.

***Algorithm Hill Climber***

The problem with the Hill climber Heuristic algorithm it usually stocks in the local optima and couldn’t

find a good solution.

**4.2 Simulated Annealing:**

***Algorithm Simulated Annealing***

*until stopping criterion is true*

Simulated Annealing is a probabilistic optimization methods that uses the idea of annealing process of thermodynamics so that it calculates the acceptance probability as follows;

Where is the temperature at step . So that if the temperature value is high enough the algorithm will choose the candidate solution even if it’s not better than the previous one when algorithm iterates though the temperature will decrease in each iteration with given temperature change ( parameter thus after some iteration if the algorithm couldn’t find a better solution then it won’t accept the candidate solution.

**4.3 TABU:**

TABU is a metaheuristic method that uses both local search paradigm and memory for optimization. In every iteration TABU calculates the neighbors of current state and choose the best one. For the next iteration as a starting point If the best neighbor is better than the best solution then it saves as a best solution. The main advantage of the TABU is it escapes the local optima by choosing best neighbor state even if has worse solution. On the other hand using memory it remembers the action that accomplishes the best neighbor state and that it uses that information to break cycling. (It doesn’t allow the action that is the reverse of the previous actions that accomplish the previous best neighbors).

***Algorithm TABU Search***

**5.0 Implementation of Metaheuristics**

**Hill Climber Heuristic:**

In our implementation of Hill Climber we use the idea of stochastic Hill Climber method. First, the algorithm chooses day, period and room randomly and acts depend on the chosen slot. If the randomly chosen slot is empty then it chooses a course randomly and calculate the new state if it assign the chosen course to chosen time slot if it’s a better solution then it select it as current state. If the given day, period room slot is not empty then algorithm tries to remove the course in given slot and calculate the value if the new value is better than it applies the remove method to the current state and saves it. The algorithm runs until the time limits is reached. Thus in each iteration algorithm selects the best action (remove or assign) then saves the action output depend on the value it produce.

Since iteration though all day, periods and room is costly for each iteration we decided to get those values by randomly. The problem of the stochastic idea is every time we run the algorithm it can generate worse solution but with the enough time limit it finds a fusible solution because it is too fast.

***Pseudo Code Stochastic Hill Climber***

*Search (*

*][room] is empty then*

*If f() >f( then*

*][room] is not empty then*

*If f() >f( then*

*return s*

**Simulated Annealing Heuristic:**

In Simulated Annealing Heuristic we have 3 different action assign a course , remove a course , swap courses each iteration the algorithm calculates the new state values accomplished by those there actions and chooses the best action in current state. After deciding the best action, it calculates the delta and if the delta is negative (which means the best action gives better result) then it runs the best action and save the new state if the delta is not negative (which means the best action gives worse solution) then it calculates the probability of the state (with current delta and temperature) and it does it runs the best action and save the new state according to the probability value. Similar to stochastic Hill Climber, Simulated Annealing chooses the day, period, room slot randomly but tries to generate new slots until it finds an action that doesn’t violate any hard constraint. The temperature is decreased at the end of each iteration. The speed of decrease is determined by the temperature change parameter.

***Pseudo Code Simulated Annealing***

*Search (*

***repeat***

*Calculate new solution by assign, remove and swap operations*

***Until at least one of the actions has no hard Constraint violations***

*choose best action m {Remove,Assign,Swap} has lowest f( )*

*return s*

**TABU:**

The algorithm iterates through all time slots and rooms, generating all possible neighbor based on the current time slot. The value change is computed for each neighbor, and the best solution encountered in the current iteration is kept in memory. The best neighbor encountered in an iteration becomes the current solution in the next iteration. To determine the possible number of neighbors, we calculate the following numbers:

d = number of days

p = number of periods per days

r = number of rooms

N(R) = number of neighbors from remove

N(S) = number of neighbors from swap

N(A) = number of neighbors from assign

N(R) = d\*p\*r (at most because program discards the day, period and room slots if it’s already empty).

N(S) = d\*(d-1)\*p\*(p-1)\*r\*(r-1) (at most because program discards the empty slots).

N(A) = d\*p\*r (at most because program discards the non-empty slots ).

Thus the maximum number of neighbors calculated in one iteration

N(T) = N(R) + N(A) + N(S) = d\*p\*r + N(S) = d\*p\*r + d\*(d-1)\*p\*(p-1)\*r\*(r-1) (if all the time slots are not empty)

After each iteration the algorithm chooses the best neighbors and save it for next iteration to start with. And if the current best neighbors has better solution than it also saves the best neighbor’s schedule into the best schedule. And add the operation that gives the best neighbor to the taboo list.

***Pseudo Code TABO***

*Search (*

*) and is not tabu*

*) and is not tabu*

*end for*

*) and is not tabu*

*end for*

*end for*

*addTaboo*

*) < )*

# **6.0 Neighborhood Function**

The neighbors of a solution are the set of different solutions, that are in some sense “next to” the solution. The concept can be interpreted geometrically by imagining all the possible solutions to our problem mapped as points in some *n*-dimensional space. The neighbors of a solution are then those points that are reachable without going through some other point.

For the university time tabling problem, a solution consists of a set of assignments consisting of a course, a time slot and a room. Each solution has an associated *value*, which is given by the objective function. To manipulate the solution, we can either *add* or *remove* a course assignment. Using these two basic operations allows us to permute any solution into any other solution. Thus we can define the set of neighbors *N(s)* for a solution *s* to be any solution that can be reached by applying exactly one basic operation to that solution, subject to the constraints given in the problem description.

Using this definition of the neighborhood, each solution has at most *time slots* × *rooms* × *courses* neighbors. The definition does have the problem that we rarely increase the solution value by removing a course, and as such this operation will be relatively infrequent (though still present) during iteration with Hill Climbing and TABU search. It seems intuitively more appealing to *swap* two scheduled courses and evaluate the resulting change in solution value. We therefore define the neighborhood of a solution to be those solutions that can be reached by *adding* or *removing* a lecture or by *swapping* two scheduled lectures.

Amending the definition of the neighborhood increases its size, and therefore the work involved in each iteration, but will hopefully yield better solutions with fewer iterations. Informal testing during development shows that this is indeed the case for the provided test data sets.

# **7.0 Delta Evaluation**

As the heuristic explores the neighborhood around a solution, it must evaluate the value of each neighbor. This can be implemented naïvely by simply running the objective function on each neighbor. This approach is correct, but will be prohibitively slow for any but the simplest problems. In our problem, the objective function will need to at least consider every room and time slot, yielding a running time of at least *O(time slots* × *rooms* × *courses)*. In addition to this, some of the penalties requires the algorithm to examine the curricula associated with each course and its neighbors.

Delta evaluation is an alternative approach that enables a faster evaluation of a proposed change to a solution. By keeping track of the solution state, we can determine the impact of a basic operation much faster than we can compute the value of an entire solution. In general, implementing delta evaluation is a tradeoff between running time improvements vs. added code complexity and increased memory consumption.

Our implementation of delta evaluation is based on the following key insights:

* *Adding* or *removing* a course lecture only affects the course itself, and possibly its neighbors due to the penalty associated with secluded lecture.
* The *swap* operation can be implemented as a series of *add* and *remove* operations.
* We must store the total solution value during execution and then adjust it when performing an operation.
* Most of the state can be efficiently maintained such that reading and updating it is a *O(1)* operation.

Our implementation uses counters and flags stored in arrays to enable constant-time lookup. Overall, computing the delta for an operation is done in *O(q)* time, with *q* being the number of curricula associated with the given course.

We use the following variables to perform delta evaluation:

// The total number of times that each course has been scheduled.  
int[] **courseAssignmentCount**;  
// The number of lectures assigned for a course on a given day.  
int[][] **courseLecturesOnDay**;  
// The number of working days for a course.  
int[] **courseWorkingDays**;  
// The number of lectures for a course scheduled in a given room.  
int[][] **lecturesInRoomForCourse**;  
// Is a lecturer busy on a given time?  
boolean[][][] **lecturerBusy**;  
// Is a given curriculum is assigned on a day and period.  
boolean[][][] **curriculumAssigned**;

Each variable is updated when an operation is performed. The ability to evaluate neighbors in near-constant time allows us to iterate orders of magnitude faster than invoking the objective function for each neighbor. The state variables above also allows us to check constraints in O(1) (see the DeltaState class for implementation of this functionality.)

**8.0 Parameter Tuning**

In this section, the report will present a brief discussion about parameter tuning, followed by the parameter tuning of Simulated Annealing and TABU.

In Parameter tuning, it will calculate average (gap) performance which is where zopt stands for the optimal value among certain results. Meanwhile, standard deviation is measured by using this formula, *stdd* = .

**Hill Climbing**

The case of Hill Climbing is the simplest, since it does not take any parameters. We simply run the algorithm 8 times for each data set to obtain the following table of statistics from each data set:

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset** | **Avg. value** | **Stdev** | **Optimal value** |
| 1 | 627 | 31 | 578 |
| 2 | 1276 | 60 | 1176 |
| 3 | 1023 | 50 | 951 |
| 4 | 1030 | 52 | 951 |
| 5 | 1312 | 38 | 1259 |
| 6 | 1717 | 52 | 1625 |
| 7 | 1962 | 57 | 1914 |
| 8 | 1298 | 82 | 1185 |
| 9 | 1102 | 69 | 1016 |
| 10 | 1566 | 53 | 1457 |
| 11 | 582 | 16 | 556 |
| 12 | 1686 | 62 | 1577 |
| 13 | 1186 | 32 | 1141 |

**Simulated Annealing**

For the case of Simulated Annealing Method, 2 parameters are used which are temperature and cooling temperature. In this project, 2 initial temperature which are 90 and 45 Celsius degree and 3 cooling temperature which are 0.95, 0.97 and 0.99 are used, making up 6 combinations. 13 different test data are tested and the final outcomes are shown in the tables below. In general, parameter with 90 as initial temperature and 0.97 as cooling temperature obtains more favourable and stable result compared to other parameters as this pair of parameters generates data with lower standard deviation and average gap. The pair (90, 0.95) works very well sometimes but it is less stable than the pair (90, 0.97).

Furthermore, Simulated Annealing with higher initial temperature produces more iteration counts and leads to lower mean value in general.

In the data analysis, green represents lower value while red represents higher value where green is favourable as the purpose of this project is to minimize the objective value.

Test 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 19.7180 | 43.9598 | 331 | 396.2667 |
| 90 | 0.97 | 11.6219 | 36.3976 | 335 | 373.9333 |
| 90 | 0.99 | 9.4131 | 69.8086 | 284 | 329.3333 |
| 45 | 0.95 | 14.8955 | 36.8631 | 367 | 421.6667 |
| 45 | 0.97 | 30.3149 | 45.0526 | 307 | 400.0667 |
| 45 | 0.99 | 12.6220 | 39.6923 | 328 | 369.4000 |

Test 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 16.2140 | 46.7668 | 567 | 658.9333 |
| 90 | 0.97 | 14.8956 | 50.7707 | 559 | 642.2667 |
| 90 | 0.99 | 11.5808 | 37.4643 | 582 | 649.4000 |
| 45 | 0.95 | 23.3987 | 67.5687 | 561 | 692.2667 |
| 45 | 0.97 | 15.3150 | 48.1427 | 582 | 671.1333 |
| 45 | 0.99 | 17.0900 | 52.2046 | 559 | 654.5333 |

Test 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 18.1360 | 43.81019668 | 397 | 469 |
| 90 | 0.97 | 18.6423 | 39.09441563 | 383 | 454.4 |
| 90 | 0.99 | 18.2317 | 37.99029167 | 377 | 445.7333 |
| 45 | 0.95 | 30.1268 | 45.9156715 | 368 | 478.8667 |
| 45 | 0.97 | 14.0000 | 30.91666217 | 410 | 467.4 |
| 45 | 0.99 | 14.8768 | 29.29345775 | 406 | 466.4 |

Test 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 19.0138 | 28.6018 | 338 | 402.2667 |
| 90 | 0.97 | 27.3354 | 40.6090 | 319 | 406.2000 |
| 90 | 0.99 | 42.3129 | 54.6624 | 294 | 418.4000 |
| 45 | 0.95 | 38.6932 | 58.1789 | 301 | 417.4667 |
| 45 | 0.97 | 25.1205 | 47.2607 | 332 | 415.4000 |
| 45 | 0.99 | 36.1876 | 45.9952 | 327 | 445.3333 |

Test 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 4.3734 | 24.9652 | 939 | 980.0667 |
| 90 | 0.97 | 3.7271 | 25.2129 | 948 | 983.3333 |
| 90 | 0.99 | 8.8294 | 35.4235 | 897 | 976.2000 |
| 45 | 0.95 | 5.8112 | 39.3077 | 943 | 997.8000 |
| 45 | 0.97 | 6.2642 | 28.7181 | 911 | 968.0667 |
| 45 | 0.99 | 8.3371 | 42.2672 | 894 | 968.5333 |

Test 6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 11.4299 | 48.1330 | 739 | 823.4667 |
| 90 | 0.97 | 7.8453 | 51.2743 | 741 | 799.1333 |
| 90 | 0.99 | 20.3016 | 75.9040 | 619 | 744.6667 |
| 45 | 0.95 | 20.8980 | 62.3222 | 683 | 825.7333 |
| 45 | 0.97 | 19.7923 | 77.4388 | 674 | 807.4000 |
| 45 | 0.99 | 28.3842 | 99.9859 | 590 | 757.4667 |

Test 7

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 14.1339 | 75.2582 | 891 | 1016.9333 |
| 90 | 0.97 | 8.3247 | 45.9066 | 965 | 1045.3333 |
| 90 | 0.99 | 5.9306 | 39.1644 | 942 | 997.8667 |
| 45 | 0.95 | 16.7943 | 77.3002 | 888 | 1037.1333 |
| 45 | 0.97 | 10.0566 | 60.6558 | 942 | 1036.7333 |
| 45 | 0.99 | 9.3467 | 58.0028 | 898 | 981.9333 |

Test 8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 20.3551 | 40.9039 | 413 | 497.0667 |
| 90 | 0.97 | 17.9831 | 44.2265 | 433 | 510.8667 |
| 90 | 0.99 | 25.6410 | 55.6880 | 403 | 506.3333 |
| 45 | 0.95 | 50.4583 | 64.6878 | 320 | 481.4667 |
| 45 | 0.97 | 24.7059 | 52.9140 | 408 | 508.8000 |
| 45 | 0.99 | 30.2302 | 48.9356 | 391 | 509.2000 |

Test 9

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 25.7436 | 31.9910 | 325 | 408.6667 |
| 90 | 0.97 | 15.0185 | 42.5527 | 360 | 414.0667 |
| 90 | 0.99 | 16.7908 | 41.0119 | 349 | 407.6000 |
| 45 | 0.95 | 18.4964 | 47.7039 | 368 | 436.0667 |
| 45 | 0.97 | 11.4392 | 26.1418 | 359 | 400.0667 |
| 45 | 0.99 | 30.7395 | 40.3911 | 311 | 406.6000 |

Test 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 10.7875 | 46.3261 | 673 | 745.6000 |
| 90 | 0.97 | 13.6170 | 46.9712 | 611 | 694.2000 |
| 90 | 0.99 | 17.5986 | 57.7232 | 583 | 685.6000 |
| 45 | 0.95 | 15.9473 | 64.7551 | 607 | 703.8000 |
| 45 | 0.97 | 8.9229 | 41.3045 | 653 | 711.2667 |
| 45 | 0.99 | 27.3345 | 55.5304 | 559 | 711.8000 |

Test 11

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 62.4422 | 29.7869 | 101 | 164.0667 |
| 90 | 0.97 | 31.5288 | 29.0894 | 133 | 174.9333 |
| 90 | 0.99 | 30.0275 | 20.4765 | 121 | 157.3333 |
| 45 | 0.95 | 28.0000 | 32.1355 | 140 | 179.2000 |
| 45 | 0.97 | 25.6808 | 23.5340 | 142 | 178.4667 |
| 45 | 0.99 | 25.7210 | 23.7303 | 141 | 177.2667 |

Test 12

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 9.7128 | 61.5873 | 917 | 1006.0667 |
| 90 | 0.97 | 5.8891 | 39.6415 | 926 | 980.5333 |
| 90 | 0.99 | 4.9658 | 36.8555 | 925 | 970.9333 |
| 45 | 0.95 | 5.9944 | 31.7658 | 965 | 1009.0667 |
| 45 | 0.97 | 8.7963 | 31.9113 | 1085 | 1018.3333 |
| 45 | 0.99 | 7.7766 | 31.5743 | 1012 | 996.9333 |

Test 13

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Temperature | Cooling Temperature | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 90 | 0.95 | 26.0668 | 46.2656 | 389 | 490.4000 |
| 90 | 0.97 | 20.3365 | 54.3774 | 416 | 500.6000 |
| 90 | 0.99 | 42.6256 | 59.9154 | 325 | 463.5333 |
| 45 | 0.95 | 34.3270 | 60.3022 | 369 | 495.6667 |
| 45 | 0.97 | 15.4705 | 38.4268 | 418 | 482.6667 |
| 45 | 0.99 | 18.0521 | 50.0582 | 397 | 468.6667 |

Graph below shows that the result is converging to the optimal value.

**TABU**

For the case of TABU, 2 parameters, taboo lengths of 5 and 20 are benchmarked. The tables below show the result of running TABU 15 times on each of the 13 test data sets.

Test 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 24.93554328 | 20.31047896 | 181 | 226.1333 |
| 5 | 24.0530303 | 42.99715753 | 176 | 218.3333 |

Test 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 30.65527066 | 67.69526489 | 468 | 611.4667 |
| 5 | 24.0530303 | 42.99715753 | 176 | 218.3333 |

Test 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 25.37286613 | 66.61217773 | 371 | 465.1333 |
| 5 | 23.69281046 | 67.01508951 | 408 | 504.6667 |

Test 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 27.24043716 | 35.13946816 | 244 | 310.4667 |
| 5 | 20.70818071 | 40.82624428 | 273 | 329.5333 |

Test 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 10.03137255 | 52.04160729 | 850 | 935.2667 |
| 5 | 11.14859438 | 67.5365251 | 830 | 922.5333 |

Test 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 33.93393393 | 135.6587221 | 444 | 594.6667 |
| 5 | 21.90895742 | 53.54981066 | 454 | 553.4667 |

Test 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 27.54593176 | 59.8592423 | 508 | 647.9333 |
| 5 | 23.91242938 | 69.13548695 | 472 | 584.8667 |

Test 8

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 33.16312057 | 59.83139273 | 235 | 312.9333 |
| 5 | 20.65934066 | 46.71587881 | 273 | 329.4 |

Test 9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 24.95575221 | 65.60975537 | 339 | 423.6 |
| 5 | 11.49911817 | 29.79679327 | 378 | 421.4667 |

Test 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 24.95575221 | 65.60975537 | 339 | 423.6 |
| 5 | 13.05691057 | 36.41403881 | 410 | 463.5333 |

Test 11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 21.50641026 | 35.17284306 | 208 | 252.7333 |
| 5 | 20.95744681 | 28.44362846 | 188 | 227.4 |

Test 12

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 12.10176991 | 92.31309044 | 904 | 1013.4 |
| 5 | 14.4191344 | 102.247934 | 878 | 1004.6 |

Test 13

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taboo Length | Average Gap Performance | Standard Deviation | Optimal | Mean |
| 20 | 33.92081737 | 58.80120936 | 261 | 349.5333 |
| 5 | 19.28057554 | 34.73480483 | 278 | 331.6 |

1. **Conclusion**

It is readily apparent that the TABU heuristic produces the best results overall. The average and optimal values are consistently lower than their Simulated Annealing and Hill Climbing counterparts. Furthermore, the values are more stable than the output of the other methods. Interestingly, the taboo list length of 5 was more efficient than a longer list. This might either be due to the longer list imposing too many restrictions neighbor exploration, or it might simply be due to the added work required to check a longer list. It would be feasible to refine our approach to enable O(1) check for taboos by using a hashing structure rather than a list.

It is interesting to consider the number of iterations performed by each heuristic. The Hill Climber algorithm will perform a number of iterations that is orders of magnitude greater than the TABU algorithm. However, its stochastic approach as well as its susceptibility to being trapped by a local optimum means that it consistently performs significantly worse than its competitors. It might be possible to alleviate this somewhat by introducing random restarts, but it is not clear that this would improve the situation significantly. Indeed, this would only be the case if the solution space of the university time tabling problem is reasonably “smooth” and without many local minima.

Given the work performed by the algorithm, it has been necessary to implement a fast and efficient evaluation of potential solutions. Our implementation of delta evaluation solves this need by evaluating a candidate solution in close to constant time. Further work could be done to improve this and achieve constant time execution in all cases.

**Appendices**

1. **Source code**

**UniversityTimetabling.java**

import com.opencsv.CSVWriter;

import java.io.FileWriter;

import java.io.Writer;

import java.util.Arrays;

/\*\*

\* Created by Martin on 16-03-2015.

\*/

public class UniversityTimeTabling {

/\*\*

\* This method is called when the program is started from the command line

\* @param args Command-line arguments

\*/

public static void main(String[] args) throws Exception {

// If we have less than 8 args, we are running in normal "non-benchmark" mode

// Simply solve the given problem and exit

if (args.length <= 8) {

// Simply solve the problem without benchmarking

new UniversityTimeTabling().startWithParameters(args);

return;

}

// Check that "benchmark" is present

if (!args[0].equals("benchmark")) {

print("Too many arguments -- use parameter benchmark <run\_count> to benchmark");

return;

}

// solve the problem several times - and output intermediate results

int iterationCount = Integer.parseInt(args[1]);

// "cut off" the first two parameters

String[] arguments = new String[args.length - 1];

Arrays.asList(args).subList(2, args.length).toArray(arguments);

Writer fi = new FileWriter("output.csv");

CSVWriter w = new CSVWriter(fi, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

// SET THIS TO THE AMOUNT OF LOGICAL PROCESSORS IN YOUR MACHINE

int threadCount = Runtime.getRuntime().availableProcessors();

System.out.println("Running " + threadCount + " simultaneous benchmarks");

int i = 0;

Thread[] threads = new Thread[threadCount];

while (i < iterationCount)

{

int j = 0;

while (j < threadCount && i < iterationCount) {

BenchmarkLauncher b = new BenchmarkLauncher();

b.arguments = arguments;

b.iteration = i;

b.writer = w;

Thread t = new Thread(b);

threads[j] = t;

t.start();

j++;

i++;

}

for (int k = 0; k < j; k++)

threads[k].join();

}

w.flush();

fi.close();

}

public boolean enableBenchmarking = false;

public CSVWriter writer = null;

public void startWithParameters(String[] args) throws Exception {

// Validate input parameter count and length

if (args == null || args.length < 7) {

print("Too few arguments -- expecting basic.utt courses.utt lecturers.utt rooms.utt curricula.utt relation.utt unavailability.utt 300");

System.exit(1);

}

// Final argument is the timeout, which is optional and defaults to 300

int timeout = 300;

if (args.length >= 8) try {

timeout = Integer.parseInt(args[7], 10);

} catch (NumberFormatException nEx) {

}

String basicFile = args[0];

String coursesFile = args[1];

String lecturersFile = args[2];

String roomsFile = args[3];

String curriculaFile = args[4];

String relationFile = args[5];

String unavailabilityFile = args[6];

// Create the heuristic

Heuristic heuristic = new SimulatedAnnealing(95,0.97); // StochasticTABU(20);

heuristic.setTimeout(timeout);

print("Using heuristic " + heuristic.getClass().getSimpleName());

print("Running for " + timeout + " seconds");

// Load basic info about the problem

debug("Loading basic info...");

BasicInfo basicInfo = new BasicInfo();

basicInfo.loadFromFile(basicFile);

heuristic.basicInfo = basicInfo;

// Load the curriculum<->course mappings

debug("Loading curricula...");

heuristic.curriculum = new Curriculum();

heuristic.curriculum.loadFromFile(curriculaFile, relationFile, basicInfo.courses);

// Load the list of lecturers

debug("Loading lecturers...");

heuristic.lecturers = new Lecturers();

heuristic.lecturers.loadFromFile(lecturersFile);

// Load the list of courses

debug("Loading courses...");

heuristic.courses = new Courses();

heuristic.courses.loadFromFile(coursesFile, basicInfo.courses);

// Load unavailability times

debug("Loading unavailability...");

heuristic.unavailability = new Unavailability();

heuristic.unavailability.loadFromFile(unavailabilityFile, basicInfo.days, basicInfo.periodsPerDay, basicInfo.courses);

// Load rooms capacity

debug("Loading rooms...");

heuristic.rooms = new Rooms();

heuristic.rooms.loadFromFile(roomsFile, basicInfo.rooms);

debug("Done loading problem definition");

debug("Generating initial solution...");

Schedule initialSchedule = heuristic.getRandomInitialSolution();

debug("Starting search...");

heuristic.deltaState.initialize(initialSchedule);

Schedule solution = heuristic.search(initialSchedule);

int objectiveValue;

debug("Calculating objective value");

objectiveValue = heuristic.evaluationFunction(solution);

print("Found a solution in " + heuristic.iterationCount + " iterations");

print("The value is " + objectiveValue);

if (!enableBenchmarking) {

print(solution.toString());

}

if (enableBenchmarking && writer != null) {

synchronized (writer) {

// TODO write results to a CSV file

String[] result = new String[]{"" + objectiveValue, heuristic.getClass().getSimpleName(), heuristic.iterationCount + ""};

writer.writeNext(result);

}

}

}

/\*\*

\* Indicates whether to include detailed logging in the output.

\*/

public static boolean enableDebugOutput = false;

/\*\*

\* Print a debug message, if debug output is enabled.

\* @param message The message to print

\*/

private static void debug(String message) {

if (enableDebugOutput)

System.out.println(message);

}

/\*\*

\* Print an output message.

\* @param message The message to print

\*/

private static void print(String message) {

System.out.println(message);

}

}

**BasicInfo.java**

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Scanner;

public class BasicInfo {

public int courses, rooms, days, periodsPerDay, curricula, constraints, lecturers;

public void loadFromFile(String file) throws FileNotFoundException {

Scanner scanner = null;

try {

scanner = new Scanner(new BufferedReader(new FileReader(file)));

// Skip the first line

scanner.nextLine();

// Read tokens from the file in the order

// Courses Rooms Days Periods\_per\_day Curricula Constraints Lecturers

courses = scanner.nextInt(10);

rooms = scanner.nextInt(10);

days = scanner.nextInt(10);

periodsPerDay = scanner.nextInt(10);

curricula = scanner.nextInt(10);

constraints = scanner.nextInt(10);

lecturers = scanner.nextInt(10);

} finally {

if (scanner != null)

scanner.close();

}

}

}

**BenchMarkLauncher.java**

import com.opencsv.CSVWriter;

import java.io.FileWriter;

import java.io.Writer;

public class BenchmarkLauncher implements Runnable {

public int iteration = 0;

public String[] arguments;

public CSVWriter writer;

@Override

public void run() {

try {

// write results to a CSV file

//Writer fi = new FileWriter("output\_" + iteration + ".csv");

//CSVWriter w = new CSVWriter(fi, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

UniversityTimeTabling t = new UniversityTimeTabling();

t.enableBenchmarking = true;

t.writer = writer;

t.startWithParameters(arguments);

} catch (Exception ex) {

System.err.println(ex.getMessage());

}

}

}

**Courses.java**

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Scanner;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Courses {

/\*\*

\* The lecturer teaching the course.

\*/

public int[] lecturerForCourse;

/\*\*

\* The minimum number of lectures that should be scheduled for this course.

\*/

public int[] numberOfLecturesForCourse;

/\*\*

\* The minimum number of distinct days that course lectures should be scheduled on.

\*/

public int[] minimumWorkingDaysForCourse;

/\*\*

\* The number of students attending this course.

\*/

public int[] numberOfStudentsForCourse;

public void loadFromFile(String file, int numberOfCourses) throws FileNotFoundException {

lecturerForCourse = new int[numberOfCourses];

numberOfLecturesForCourse = new int[numberOfCourses];

minimumWorkingDaysForCourse = new int[numberOfCourses];

numberOfStudentsForCourse = new int[numberOfCourses];

Pattern linePattern = Pattern.compile("C(\\d+) L(\\d+) (\\d+) (\\d+) (\\d+)");

Scanner scanner = null;

try {

scanner = new Scanner(new BufferedReader(new FileReader(file)));

// Skip the first line

scanner.nextLine();

while (scanner.hasNextLine()) {

// Read tokens from the file in the order

// Course Lecturer Number\_of\_lectures Minimum\_working\_days Number\_of\_students

Matcher match = linePattern.matcher(scanner.nextLine());

if (!match.matches())

continue;

int courseId = Integer.parseInt(match.group(1));

lecturerForCourse[courseId] = Integer.parseInt(match.group(2));

numberOfLecturesForCourse[courseId] = Integer.parseInt(match.group(3));

minimumWorkingDaysForCourse[courseId] = Integer.parseInt(match.group(4));

numberOfStudentsForCourse[courseId] = Integer.parseInt(match.group(5));

}

} finally {

if (scanner != null)

scanner.close();

}

}

}

**Curriculum.java**

import java.io.\*;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

/\*\*

\* Created by Martin on 16-03-2015.

\*/

public class Curriculum {

/\*\*

\* Look up whether there is a relation between [course][curriculum]

\*/

boolean isCourseInCurriculum[][];

/\*\*

\* Contains a list of curricula for each course.

\*/

ArrayList<LinkedList<Integer>> curriculaForCourse;

public void loadFromFile(String curriculaFile, String relationFile, int numberOfCourses) {

// Load the list of curricula

try {

// Parse the curricula file

// Each line has the curriculum number and course count

Reader reader = new FileReader(curriculaFile);

BufferedReader bufferedReader = new BufferedReader(reader);

Pattern p = Pattern.compile("Q(\\d{4}) (\\d)");

String latestLine;

int curriculaCount = 0;

while ((latestLine = bufferedReader.readLine()) != null) {

Matcher m = p.matcher(latestLine);

if (!m.matches())

continue;

// Index of the curriculum we just parsed

int curriculumIndex = Integer.parseInt(m.group(1));

// The number of curricula is always equal to highest index + 1

curriculaCount = 1 + curriculumIndex;

}

bufferedReader.close();

reader.close();

curriculaForCourse = new ArrayList<LinkedList<Integer>>(numberOfCourses);

for (int i = 0; i < numberOfCourses; i++)

curriculaForCourse.add(new LinkedList<Integer>());

// Store assignments as an array of booleans

isCourseInCurriculum = new boolean[numberOfCourses][curriculaCount];

// Parse the relation file

// Each line has the index of a curriculum and a course

Pattern relationPattern = Pattern.compile("Q(\\d{4}) C(\\d{4})");

reader = new FileReader(relationFile);

bufferedReader = new BufferedReader(reader);

while ((latestLine = bufferedReader.readLine()) != null) {

Matcher m = relationPattern.matcher(latestLine);

if (!m.matches())

continue;

// Indices we just read from the file

int curriculumId = Integer.parseInt(m.group(1));

int courseId = Integer.parseInt(m.group(2));

isCourseInCurriculum[courseId][curriculumId] = true;

curriculaForCourse.get(courseId).add(curriculumId);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Lecturers.java**

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.\*;

public class Lecturers {

public int[] lecturers;

public void loadFromFile(String file) throws FileNotFoundException {

Scanner scanner = null;

List idList = new LinkedList<Integer>();

try {

scanner = new Scanner(new BufferedReader(new FileReader(file)));

// Skip the first line

scanner.nextLine();

//scanner.useDelimiter("\\s\*L");

// Read tokens from the file

// Each line contains an id

System.out.println("Starting scan...");

String pattern = "L\\d+";

while (scanner.hasNext(pattern)) {

String line = scanner.next(pattern);

int id = Integer.parseInt(line.substring(1));

idList.add(id);

}

lecturers = new int[idList.size()];

Iterator<Integer> iter = idList.iterator();

for (int i=0; iter.hasNext(); i++) {

lecturers[i] = iter.next();

}

} finally {

if (scanner != null)

scanner.close();

}

}

}

**Rooms.java**

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Scanner;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Rooms {

/\*\*

\* Look up whether a course is unavailable on [day][period][course]

\*/

public int[] capacityForRoom;

public void loadFromFile(String file, int numberOfRooms) throws FileNotFoundException {

capacityForRoom = new int[numberOfRooms];

Pattern linePattern = Pattern.compile("R(\\d+) (\\d+)");

Scanner scanner = null;

try {

scanner = new Scanner(new BufferedReader(new FileReader(file)));

// Skip the first line

scanner.nextLine();

while (scanner.hasNextLine()) {

// Read tokens from the file in the order

// Course Day Period

Matcher match = linePattern.matcher(scanner.nextLine());

if (!match.matches())

continue;

int room = Integer.parseInt(match.group(1));

int capacity = Integer.parseInt(match.group(2));

capacityForRoom[room] = capacity;

}

} finally {

if (scanner != null)

scanner.close();

}

}

}

**Schedule.java**

public class Schedule {

public int[][][] assignments; //The array of scheduled courses, look up using [day][period][room],Value is -1 if the room is empty

public Schedule(int days, int periods, int rooms) {

assignments = new int[days][periods][rooms];

// initialize all assignments to -1, meaning no course

for (int d = 0; d < assignments.length; d++) {

for (int p = 0; p < assignments[d].length; p++) {

for (int r = 0; r < assignments[d][p].length; r++) {

assignments[d][p][r] = -1;

}

}

}

}

@Override

public String toString() {

StringBuilder builder = new StringBuilder();

for (int d = 0; d < assignments.length; d++) {

for (int p = 0; p < assignments[d].length; p++) {

for (int r = 0; r < assignments[d][p].length; r++) {

int course = assignments[d][p][r];

if (course == -1)

continue;

builder.append(String.format("C%04d %d %d R%04d\n", course, d, p, r));

}

}

}

return builder.toString();

}

}

**Unavailability.java**

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Unavailability {

/\*\*

\* Look up whether a course is unavailable on [day][period][course]

\*/

public boolean[][][] courseUnavailable;

public List<UnavailabilityConstraint> constraints;

public void loadFromFile(String file, int numberOfDays, int numberOfPeriods, int numberOfCourses) throws FileNotFoundException {

courseUnavailable = new boolean[numberOfDays][numberOfPeriods][numberOfCourses];

constraints = new LinkedList<UnavailabilityConstraint>();

Pattern linePattern = Pattern.compile("C(\\d+) (\\d+) (\\d+)");

Scanner scanner = null;

try {

scanner = new Scanner(new BufferedReader(new FileReader(file)));

// Skip the first line

scanner.nextLine();

while (scanner.hasNextLine()) {

// Read tokens from the file in the order

// Course Day Period

Matcher match = linePattern.matcher(scanner.nextLine());

if (!match.matches())

continue;

UnavailabilityConstraint constraint = new UnavailabilityConstraint();

constraint.course = Integer.parseInt(match.group(1));

constraint.day = Integer.parseInt(match.group(2));

constraint.period = Integer.parseInt(match.group(3));

constraints.add(constraint);

courseUnavailable[constraint.day][constraint.period][constraint.course] = true;

}

} finally {

if (scanner != null)

scanner.close();

}

}

}

**UnavailabilityConstraint.java**

public class UnavailabilityConstraint {

public int day, period, course;

}

**XORShiftRandom.java**

import java.util.Random;

/\*\*

\* A faster, higher-quality replacement for java.util.Random.

\* See http://www.javamex.com/tutorials/random\_numbers/xorshift.shtml

\*/

public class XORShiftRandom extends Random {

private long seed = System.nanoTime();

public XORShiftRandom() {

}

protected int next(int nbits) {

// N.B. Not thread-safe!

long x = this.seed;

x ^= (x << 21);

x ^= (x >>> 35);

x ^= (x << 4);

this.seed = x;

x &= ((1L << nbits) -1);

return (int) x;

}

}

**Heuristic.java**

import java.io.IOException;

import java.io.Writer;

import java.util.Date;

import java.util.LinkedList;

import java.util.Random;

import com.opencsv.CSVWriter;

public abstract class Heuristic {

public static final int EMPTY\_ROOM = -1;

/\*\*

\* Perform a search with the input schedule as a starting point. The method will return the best schedule

\* found before timeout.

\* This method is abstract and must be implemented according to the chosen heuristic.

\* @param schedule The Schedule to start from

\* @return The most optimal schedule found during search.

\* @throws IOException

\*/

public abstract Schedule search(Schedule schedule) throws IOException;

private int timeout = 300;

/\*\*

\* Sets the duration, in seconds, that the heuristic is allowed to search for solutions.

\* @param seconds the duration in seconds. Must be a positive value.

\*/

public void setTimeout(int seconds) {

if (seconds <= 0)

return;

timeout = seconds;

}

/\*\*

\* Clones an array by copying its contents to an existing destination array.

\*/

protected static void cloneArray(int[][][] array, int[][][] destination) {

for (int i = 0; i < array.length; i++) {

for (int j = 0; j < array[i].length; j++) {

System.arraycopy(array[i][j], 0, destination[i][j], 0,

array[i][j].length);

}

}

}

/\*\*

\* The millisecond time when the countdown was started.

\*/

private long countdownStartTime;

/\*\*

\* Starts the countdown timer. Call timeoutReached() to determine when the timeout has been reached.

\*/

protected void startCountdown() {

countdownStartTime = new Date().getTime();

}

/\*\*

\* Returns a value indicating whether the timeout has currently been reached.

\* startCountdown() must be called prior to calling this method.

\* @return true if the timeout has been reached, or if the countdown has not yet started. If not, false.

\*/

protected boolean timeoutReached() {

Date currentTime = new Date();

long delta = (currentTime.getTime() - countdownStartTime) / 1000;

return delta > timeout;

}

public int iterationCount = 0;

public BasicInfo basicInfo;

public Curriculum curriculum;

public Lecturers lecturers;

public Courses courses;

public Unavailability unavailability;

public Rooms rooms;

/\*\*

\* Returns an array containing the number of times each course has been scheduled

\*/

protected int[] getCourseAssignmentCount(Schedule schedule) {

int[] result = new int[basicInfo.courses];

for (int day = 0; day < basicInfo.days; day++) {

for (int period = 0; period < basicInfo.periodsPerDay; period++) {

for (int room = 0; room < basicInfo.rooms; room++){

int assignedCourse = schedule.assignments[day][period][room];

if(assignedCourse == -1)

continue;

result[assignedCourse]++;

}

}

}

return result;

}

private Random rand = new XORShiftRandom();

public Schedule getRandomInitialSolution(){

Schedule result = new Schedule(basicInfo.days, basicInfo.periodsPerDay, basicInfo.rooms);

int[] courseAssignmentCount = new int[basicInfo.courses];

for (int day = 0; day < basicInfo.days; day++) {

boolean[] courseAlreadyAssigned = new boolean[basicInfo.courses];

for (int period = 0; period < basicInfo.periodsPerDay; period++) {

boolean[] lecturerBusy = new boolean[basicInfo.lecturers];

boolean[] curriculumBusy = new boolean[basicInfo.curricula];

for (int room = 0; room < basicInfo.rooms; room++) {

int assignedCourse = -1; // fix

int candidatecourse = -1;

int attempts = 0;

while (assignedCourse == -1) {

candidatecourse = rand.nextInt(basicInfo.courses); // maybe use a priority queue instead?

attempts++;

if (attempts == basicInfo.courses \* 2)

break;

// course not already assigned in time slot

if (courseAlreadyAssigned[candidatecourse])

continue;

// course not unavailable in time slot

if (unavailability.courseUnavailable[day][period][candidatecourse])

continue;

// course lecturer cannot be busy

if (lecturerBusy[courses.lecturerForCourse[candidatecourse]])

continue;

// course curriculum cannot be busy

boolean curriculumConflict = false;

for (int curriculum = 0; curriculum < basicInfo.curricula; curriculum++) {

if (this.curriculum.isCourseInCurriculum[candidatecourse][curriculum]) {

if (curriculumBusy[curriculum])

curriculumConflict = true;

}

}

if (curriculumConflict)

continue;

// course max lectures cannot be reached

if (courseAssignmentCount[candidatecourse] == courses.numberOfLecturesForCourse[candidatecourse])

continue;

// no constraints violated! assign this course

assignedCourse = candidatecourse;

}

if (assignedCourse == -1)

continue;

// increment constraints

courseAlreadyAssigned[assignedCourse] = true;

lecturerBusy[courses.lecturerForCourse[candidatecourse]] = true;

for (int curriculum = 0; curriculum < basicInfo.curricula; curriculum++) {

if (this.curriculum.isCourseInCurriculum[assignedCourse][curriculum]) {

curriculumBusy[curriculum] = true;

}

}

courseAssignmentCount[candidatecourse]++;

result.assignments[day][period][room] = assignedCourse;

}

}

}

return result;

}

public int evaluationFunction(Schedule schedule)

{

int[] numberOfLecturesOfCourse = new int[basicInfo.courses];

System.arraycopy(courses.numberOfLecturesForCourse, 0, numberOfLecturesOfCourse, 0, basicInfo.courses);

//to calculate the number of unallocated lectures of each course

for(int day= 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int room = 0; room < basicInfo.rooms; room++){

int assignedCourse = schedule.assignments[day][period][room];

if(assignedCourse == -1)

continue;

numberOfLecturesOfCourse[assignedCourse]--;

}

}

}

//to calculate the number of days of each course that is scheduled below the minimum number of working days

int[] minimumWorkingDaysOfCourse = new int[basicInfo.courses];

System.arraycopy(courses.minimumWorkingDaysForCourse, 0, minimumWorkingDaysOfCourse, 0, basicInfo.courses);

for(int day = 0; day < basicInfo.days; day++){

// to avoid overcount working days of each course

boolean[] dayFulfilled = new boolean[basicInfo.courses];

for(int i = 0; i < basicInfo.courses; i++){

dayFulfilled[i] = false;

}

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int room = 0; room < basicInfo.rooms; room++){

int assignedCourse = schedule.assignments[day][period][room];

if(assignedCourse == EMPTY\_ROOM)

continue;

if(dayFulfilled[assignedCourse])

continue;

dayFulfilled[assignedCourse] = true;

minimumWorkingDaysOfCourse[assignedCourse]--;

}

}

}

int[][][] secludedLecture = new int[basicInfo.days][basicInfo.periodsPerDay][basicInfo.curricula];

//Initialise the value of each slot in secludedLecture, 1 if a curriculum in a timeslot has a secluded lecture

for(int day = 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int curriculum = 0; curriculum < basicInfo.curricula; curriculum++){

secludedLecture[day][period][curriculum] = 0;

}

}

}

//to calculate the number of secluded lectures of each curriculum

for(int day = 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int curriculum = 0; curriculum < basicInfo.curricula; curriculum++){

int count1 = 0; // to calculate X\_c,t,r

int count2 = 0; // to calculate X\_c',t',r'

for(int course = 0; course < basicInfo.courses; course++){

if (this.curriculum.isCourseInCurriculum[course][curriculum]){

for(int room = 0; room < basicInfo.rooms; room++){

if(schedule.assignments[day][period][room] == course)

count1++;

}

}

}

int adjacentPeriod1 = period - 1;

int adjacentPeriod2 = period + 1;

if(adjacentPeriod1 >= 0){

for(int course = 0;course < basicInfo.courses; course++){

if(this.curriculum.isCourseInCurriculum[course][curriculum]){

for(int room = 0; room < basicInfo.rooms;room++){

if(schedule.assignments[day][adjacentPeriod1][room] == course)

count2++;

}

}

}

}

if(adjacentPeriod2 < basicInfo.periodsPerDay){

for(int course = 0;course < basicInfo.courses; course++){

if(this.curriculum.isCourseInCurriculum[course][curriculum]){

for(int room = 0; room < basicInfo.rooms;room++){

if(schedule.assignments[day][adjacentPeriod2][room] == course)

count2++;

}

}

}

}

if(count1 == 1 && count2 == 0)

secludedLecture[day][period][curriculum] = 1;

}

}

}

//to calculate number of room changes of each courses

int[] numberOfRoomChanges = new int[basicInfo.courses];

//if the course is always taught in same room, the value is 0

//if the course is never allocated, the value is -1

for(int course = 0; course < basicInfo.courses; course++){

numberOfRoomChanges[course] = -1;

}

for(int course = 0; course < basicInfo.courses; course++){

boolean[] roomChanged = new boolean[basicInfo.rooms];

for(int room = 0; room < basicInfo.rooms; room++){

roomChanged[room] = false;

}

for(int day = 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int room = 0; room < basicInfo.rooms; room++){

if(schedule.assignments[day][period][room] == course)

roomChanged[room] = true;

}

}

}

for(int room = 0; room < basicInfo.rooms; room++){

if(roomChanged[room])

numberOfRoomChanges[course]++;

}

}

//to calculate the amount of capacity that room is exceeded in a timeslot

int capacityExceeding = 0;

for(int day = 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int room = 0; room < basicInfo.rooms; room++){

int course = schedule.assignments[day][period][room];

if(course != -1){

if(this.rooms.capacityForRoom[room] < this.courses.numberOfStudentsForCourse[course])

capacityExceeding += (this.courses.numberOfStudentsForCourse[course] - this.rooms.capacityForRoom[room]);

}

}

}

}

int objective; // calculate the penalties

int unscheduled = 0;

int minimumWorkingDays = 0;

int curriculumCompactness = 0;

int roomStability = 0;

for(int i = 0; i < basicInfo.courses; i++){

unscheduled += numberOfLecturesOfCourse[i];

}

for(int course = 0; course < basicInfo.courses; course++){

if(minimumWorkingDaysOfCourse[course] > 0)

minimumWorkingDays += minimumWorkingDaysOfCourse[course];

}

for(int day = 0; day < basicInfo.days; day++){

for(int period = 0; period < basicInfo.periodsPerDay; period++){

for(int curriculum = 0; curriculum < basicInfo.curricula; curriculum++){

curriculumCompactness += secludedLecture[day][period][curriculum];

}

}

}

for(int course = 0; course < basicInfo.courses; course++){

if(numberOfRoomChanges[course] > 0)

roomStability += numberOfRoomChanges[course];

}

objective = 10\*unscheduled + 5\*minimumWorkingDays + 2\*curriculumCompactness + roomStability + capacityExceeding;

return objective;

}

/\*\*

\* Gets the value of the solution if it is altered by assigning a specific course in a given time slot and room.

\* This method return Integer.MAX\_VALUE if the room is already occupied.

\* @return The value of the modified solution, or Integer.MAX\_VALUE if a constraint is violated.

\*/

protected int valueIfAssigningCourse(Schedule schedule, int currentValue, int day, int period, int room, int courseId) {

// Room must currently be empty

if (schedule.assignments[day][period][room] != Heuristic.EMPTY\_ROOM) {

return Integer.MAX\_VALUE;

}

// Ensure that the move is valid

boolean moveIsValid = deltaState.deltaValidateAllConstraints(day, period, courseId);

if (!moveIsValid) {

// Return a large value to indicate that the move is invalid

return Integer.MAX\_VALUE;

}

int delta = deltaState.getDeltaWhenAdding(day, period, room, courseId);

return currentValue + delta;

}

/\*\*

\* Gets the value of the solution if the given time slot and room is emptied

\* @return The value of the modified solution, or Integer.MAX\_VALUE if a constraint is violated

\*/

protected int valueIfRemovingCourse(Schedule schedule, int currentValue, int day, int period, int room) {

// Room must currently be occupied

int currentCourse = schedule.assignments[day][period][room];

if (currentCourse == Heuristic.EMPTY\_ROOM) {

return Integer.MAX\_VALUE;

}

// Return the delta value plus the current value

int delta = deltaState.getDeltaWhenRemoving(day, period, room, currentCourse);

return currentValue + delta;

}

/\*\*

\* Gets the value of the solution if two lectures are swapped

\* @return The value of the modified solution, or Integer.MAX\_VALUE if a constraint is violated

\*/

protected int valueIfSwappingCourses(Schedule schedule, int currentValue, int day, int period,int room,int day2,int period2,int room2) {

// Both rooms must currently be occupied

int currentCourse = schedule.assignments[day][period][room];

int currentCourse2 = schedule.assignments[day2][period2][room2];

if (currentCourse == EMPTY\_ROOM || currentCourse2 == EMPTY\_ROOM)

return Integer.MAX\_VALUE;

int totalDelta = 0;

// First clear both rooms -- this does not require constraint validation

totalDelta += deltaState.getDeltaWhenRemoving(day, period, room, currentCourse);

removeCourse(schedule, day, period, room);

totalDelta += deltaState.getDeltaWhenRemoving(day2, period2, room2, currentCourse2);

removeCourse(schedule, day2, period2, room2);

// Then check the delta values if adding the swapped courses

boolean constraintsSatisfied;

totalDelta += deltaState.getDeltaWhenAdding(day2, period2, room2, currentCourse);

constraintsSatisfied = deltaState.deltaValidateAllConstraints(day2, period2, currentCourse);

// Assign the course so we can compute the delta

// Only assign course in slot 2

assignCourse(schedule, day2, period2, room2, currentCourse);

if (constraintsSatisfied) {

// Only compute the second delta if the first constraint is satistified

totalDelta += deltaState.getDeltaWhenAdding(day, period, room, currentCourse2);

constraintsSatisfied = deltaState.deltaValidateAllConstraints(day, period, currentCourse2);

// It is not necessary to actually assign the course in slot 1

}

// Only course assigned at this point is time slot 2

// Revert the changes by reassigning the courses. Then return the computed value.

// Remove course in time slot 2, so both slots are empty

removeCourse(schedule, day2, period2, room2);

// An reassign both courses

assignCourse(schedule, day, period, room, currentCourse);

assignCourse(schedule, day2, period2, room2, currentCourse2);

if (!constraintsSatisfied) {

return Integer.MAX\_VALUE;

}

return currentValue + totalDelta;

}

/\*\*

\* Assigns the given course to the given room and time slot.

\* If the room is already occupied, this method does nothing.

\*/

protected void assignCourse(Schedule schedule, int day, int period, int room, int course) {

// Make sure the room is empty

removeCourse(schedule, day, period, room);

if(course == Heuristic.EMPTY\_ROOM)

return;

// Perform the assignment and increment the counter

schedule.assignments[day][period][room] = course;

deltaState.courseAssignmentCount[course]++;

int lecturesOnDay = ++deltaState.courseLecturesOnDay[course][day];

if (lecturesOnDay == 1)

deltaState.courseWorkingDays[course]++;

deltaState.lecturesInRoomForCourse[room][course]++;

for (int q : curriculum.curriculaForCourse.get(course)) {

deltaState.curriculumAssigned[q][day][period]++;

}

deltaState.lecturerBusy[courses.lecturerForCourse[course]][day][period]++;

}

/\*\*

\* Empties the given room in the given time slot.

\* If the room is already empty, this method does nothing.

\*/

protected void removeCourse(Schedule schedule, int day, int period, int room) {

int assignedCourse = schedule.assignments[day][period][room];

if (assignedCourse == Heuristic.EMPTY\_ROOM)

return;

// Perform the assignment and decrement the counter

schedule.assignments[day][period][room] = EMPTY\_ROOM;

deltaState.courseAssignmentCount[assignedCourse]--;

int lecturesOnDay = --deltaState.courseLecturesOnDay[assignedCourse][day];

if (lecturesOnDay == 0)

deltaState.courseWorkingDays[assignedCourse]--;

deltaState.lecturesInRoomForCourse[room][assignedCourse]--;

for (int q : curriculum.curriculaForCourse.get(assignedCourse)) {

deltaState.curriculumAssigned[q][day][period]--;

}

deltaState.lecturerBusy[courses.lecturerForCourse[assignedCourse]][day][period]--;

}

protected enum Type { REMOVE, ASSIGN, SWAP, NOTHING }

protected CSVWriter writer = null;

protected Writer f;

protected DeltaEvaluationState deltaState = new DeltaEvaluationState();

protected class DeltaEvaluationState {

final int UNSCHEDULED\_PENALTY = 10;

final int ROOM\_CAPACITY\_PENALTY = 1; // per student

final int MINIMUM\_WORKING\_DAYS\_PENALTY = 5;

final int CURRICULUM\_COMPACTNESS\_PENALTY = 2;

/\*\*

\* Get the change in Unscheduled penalty if a lecture is assigned in the given course.

\*/

private int getUnscheduledPenaltyAfterAdding(int course) {

// Get the current number of lectures, and the minimum number allowed

int assignments = courseAssignmentCount[course];

int minimumAssignments = courses.numberOfLecturesForCourse[course];

// If we are already at the minimum, adding an extra assignment does not change the solution value

if (assignments >= minimumAssignments)

return 0;

// But in other cases the solution value is reduced by one UNSCHEDULED\_PENALTY

return -UNSCHEDULED\_PENALTY;

}

/\*\*

\* Get the change in Unscheduled penalty if one lecture of the given course is removed.

\*/

private int getUnscheduledPenaltyAfterRemoving(int course) {

// Get the current number of lectures, and the minimum number allowed

int assignments = courseAssignmentCount[course];

int minimumAssignments = courses.numberOfLecturesForCourse[course];

// If we are already above the minimum, adding an extra lecture does not change the solution value

if (assignments > minimumAssignments)

return 0;

// But in other cases the solution value is increased by one UNSCHEDULED\_PENALTY

return UNSCHEDULED\_PENALTY;

}

/\*\*

\* Gets the change in RoomCapacity penalty if one lecture in the given course is scheduled in the given room.

\*/

private int roomCapacityPenaltyAfterAdding(int room, int course) {

int numberOfStudents = courses.numberOfStudentsForCourse[course];

int roomCapacity = rooms.capacityForRoom[room];

return Math.max((numberOfStudents - roomCapacity) \* ROOM\_CAPACITY\_PENALTY, 0);

}

/\*\*

\* Gets the change in RoomCapacity penalty if one lecture in the given course is removed from the given room.

\*/

private int roomCapacityPenaltyAfterRemoving(int room, int course) {

return -roomCapacityPenaltyAfterAdding(room, course);

}

/\*\*

\* The total number of times that each course has been scheduled.

\*/

public int[] courseAssignmentCount;

private int[][] courseLecturesOnDay;

private int[] courseWorkingDays;

public int[][] lecturesInRoomForCourse;

/\*\*

\* Look up whether a lecturer is busy on a give time using [lecturer][day][period]

\*/

public byte[][][] lecturerBusy;

/\*\*

\* Look up whether a given curriculum is assigned on a day and period.

\* Use indexing [curriculum][day][period]

\*/

private byte[][][] curriculumAssigned;

public void initialize(Schedule schedule) {

curriculumAssigned = new byte[basicInfo.curricula][basicInfo.days][basicInfo.periodsPerDay];

courseAssignmentCount = getCourseAssignmentCount(schedule);

courseLecturesOnDay = new int[basicInfo.courses][basicInfo.days];

courseWorkingDays = new int[basicInfo.courses];

lecturesInRoomForCourse = new int[basicInfo.rooms][basicInfo.courses];

lecturerBusy = new byte[basicInfo.lecturers][basicInfo.days][basicInfo.periodsPerDay];

for (int day = 0; day < basicInfo.days; day++) {

for (int period = 0; period < basicInfo.periodsPerDay; period++) {

for (int room = 0; room < basicInfo.rooms; room++) {

int course = schedule.assignments[day][period][room];

if (course == EMPTY\_ROOM)

continue;

courseLecturesOnDay[course][day]++;

lecturesInRoomForCourse[room][course]++;

int lecturerId = courses.lecturerForCourse[course];

lecturerBusy[lecturerId][day][period]++;

assert lecturerBusy[lecturerId][day][period] == 1;

for (int q : curriculum.curriculaForCourse.get(course)) {

curriculumAssigned[q][day][period] = 1;

}

}

}

}

for (int course = 0; course < basicInfo.courses; course++) {

for (int day = 0; day < basicInfo.days; day++) {

if (courseLecturesOnDay[course][day] > 0)

courseWorkingDays[course]++;

}

}

}

/\*\*

\* Gets the change in MinimumWorkingDays penalty by scheduling one lecture in the given course on the given day.

\*/

public int minWorkingDaysPenaltyAfterAdding(int day, int course) {

// What is the number of course lectures scheduled on this day?

int lecturesScheduled = courseLecturesOnDay[course][day];

// If the course is already scheduled on this day, there is no change

// So the delta is 0

if (lecturesScheduled > 0)

return 0;

int minimumWorkingDays = courses.minimumWorkingDaysForCourse[course];

int currentWorkingDays = courseWorkingDays[course];

// If we are already at a sufficient number of working days, adding one does not reduce solution value

if (minimumWorkingDays <= currentWorkingDays)

return 0;

// Decreasing the working days deficit will reduce the penalty

return -MINIMUM\_WORKING\_DAYS\_PENALTY;

}

public int minWorkingDaysPenaltyAfterRemoving(int day, int course) {

// What is the number of course lectures scheduled on this day?

int lecturesScheduled = courseLecturesOnDay[course][day];

// If the course is already scheduled at least twice on this day, there is no change

// So the delta is 0

if (lecturesScheduled > 1)

return 0;

int minimumWorkingDays = courses.minimumWorkingDaysForCourse[course];

int currentWorkingDays = courseWorkingDays[course];

// If we are already at a sufficient number of working days, removing one does not increase solution value

if (minimumWorkingDays < currentWorkingDays)

return 0;

// Increasing the working days deficit will increase the penalty

return MINIMUM\_WORKING\_DAYS\_PENALTY;

}

public int roomStabilityPenaltyAfterAdding(int room, int course) {

// If this is the first lecture for the course, or the room is already in use by the course,

// there is no change -- return 0

if (courseAssignmentCount[course] == 0 || lecturesInRoomForCourse[room][course] > 0)

return 0;

// If we get to this point, the change means an extra room in use

return 1;

}

public int roomStabilityPenaltyAfterRemoving(int room, int course) {

// If this is the first lecture for the course, or the room is already in use by the course

// for several lectures, there is no change -- return 0

if (courseAssignmentCount[course] == 1 || lecturesInRoomForCourse[room][course] > 1)

return 0;

// If we get to this point, the change means an extra room in use

return -1;

}

public int curriculumCompactnessPenaltyAfterAdding(int day, int period, int course) {

LinkedList<Integer> curriculaForCourse = curriculum.curriculaForCourse.get(course);

int ownSeclusionPenalty = 0;

int otherSeclusionPenalty = 0;

for (int curriculum : curriculaForCourse) {

// Determine whether the added course is secluded with respect to this curriculum

boolean hasNeighbor = false;

boolean earlierNeighorIsSecluded = false;

boolean hasNeighborEarlier = false;

boolean hasNeighborLater = false;

boolean laterNeighborIsSecluded = false;

if (period > 0) {

hasNeighborEarlier = curriculumAssigned[curriculum][day][period - 1] > 0;

if (hasNeighborEarlier) {

// We have an earlier neighbor -- compute whether that neighbor already had a neighbor

earlierNeighorIsSecluded = period < 2 || curriculumAssigned[curriculum][day][period - 2] == 0;

}

hasNeighbor = hasNeighborEarlier;

}

if (period + 1 < basicInfo.periodsPerDay) {

hasNeighborLater = curriculumAssigned[curriculum][day][period + 1] > 0;

if (hasNeighborLater) {

// We have a later neighbor -- compute whether that neighbor was already secluded

laterNeighborIsSecluded = period > basicInfo.periodsPerDay - 3 || curriculumAssigned[curriculum][day][period + 2] == 0;

}

hasNeighbor = hasNeighbor || hasNeighborLater;

}

if (!hasNeighbor)

ownSeclusionPenalty += CURRICULUM\_COMPACTNESS\_PENALTY;

if (hasNeighborEarlier && earlierNeighorIsSecluded)

otherSeclusionPenalty -= CURRICULUM\_COMPACTNESS\_PENALTY;

if (hasNeighborLater && laterNeighborIsSecluded)

otherSeclusionPenalty -= CURRICULUM\_COMPACTNESS\_PENALTY;

}

return ownSeclusionPenalty + otherSeclusionPenalty;

}

public int curriculumCompactnessPenaltyAfterRemoving(int day, int period, int course) {

LinkedList<Integer> curriculaForCourse = curriculum.curriculaForCourse.get(course);

int ownSeclusionPenalty = 0;

int otherSeclusionPenalty = 0;

for (int curriculum : curriculaForCourse) {

// Determine whether the added course is secluded with respect to this curriculum

boolean hasNeighbor = false;

boolean earlierNeighorIsSecluded = false;

boolean hasNeighborEarlier = false;

boolean hasNeighborLater = false;

boolean laterNeighborIsSecluded = false;

if (period > 0) {

hasNeighborEarlier = curriculumAssigned[curriculum][day][period - 1] > 0;

if (hasNeighborEarlier) {

// We have an earlier neighbor -- compute whether that neighbor already had a neighbor

earlierNeighorIsSecluded = period < 2 || curriculumAssigned[curriculum][day][period - 2] == 0;

}

hasNeighbor = hasNeighborEarlier;

}

if (period + 1 < basicInfo.periodsPerDay) {

hasNeighborLater = curriculumAssigned[curriculum][day][period + 1] > 0;

if (hasNeighborLater) {

// We have a later neighbor -- compute whether that neighbor was already secluded

laterNeighborIsSecluded = period > basicInfo.periodsPerDay - 3 || curriculumAssigned[curriculum][day][period + 2] == 0;

}

hasNeighbor = hasNeighbor || hasNeighborLater;

}

if (!hasNeighbor)

ownSeclusionPenalty -= CURRICULUM\_COMPACTNESS\_PENALTY;

if (hasNeighborEarlier && earlierNeighorIsSecluded)

otherSeclusionPenalty += CURRICULUM\_COMPACTNESS\_PENALTY;

if (hasNeighborLater && laterNeighborIsSecluded)

otherSeclusionPenalty += CURRICULUM\_COMPACTNESS\_PENALTY;

}

return ownSeclusionPenalty + otherSeclusionPenalty;

}

public int getDeltaWhenAdding(int day, int period, int room, int course) {

int penaltyDelta = 0;

// Unscheduled

penaltyDelta += getUnscheduledPenaltyAfterAdding(course);

// RoomCapacity

penaltyDelta += roomCapacityPenaltyAfterAdding(room, course);

// MinimumWorkingDays

penaltyDelta += minWorkingDaysPenaltyAfterAdding(day, course);

// CurriculumCompactness

penaltyDelta += curriculumCompactnessPenaltyAfterAdding(day, period, course);

// RoomStability

penaltyDelta += roomStabilityPenaltyAfterAdding(room, course);

return penaltyDelta;

}

public int getDeltaWhenRemoving(int day, int period, int room, int course) {

int penaltyDelta = 0;

// Unscheduled

penaltyDelta += getUnscheduledPenaltyAfterRemoving(course);

// RoomCapacity

penaltyDelta += roomCapacityPenaltyAfterRemoving(room, course);

// MinimumWorkingDays

penaltyDelta += minWorkingDaysPenaltyAfterRemoving(day, course);

// CurriculumCompactness

penaltyDelta += curriculumCompactnessPenaltyAfterRemoving(day, period, course);

// RoomStability

penaltyDelta += roomStabilityPenaltyAfterRemoving(room, course);

return penaltyDelta;

}

/\*\*

\* Checks that a given course can be added in the given time slot without violating the SameCurriculum constraint.

\* @return true if the constraint is satisfied

\*/

public boolean deltaValidateSameCurriculumConstraint(int day, int period, int course) {

LinkedList<Integer> curricula = curriculum.curriculaForCourse.get(course);

for (int curriculum : curricula) {

if (curriculumAssigned[curriculum][day][period] > 0)

return false;

}

return true;

}

/\*\*

\* Checks that a given course can be added in the given time slot without violating the SameLecturer constraint.

\* @return true if the constraint is satisfied

\*/

public boolean deltaValidateSameLecturerConstraint(int day, int period, int course) {

int lecturer = courses.lecturerForCourse[course];

return lecturerBusy[lecturer][day][period] == 0;

}

/\*\*

\* Checks that a given course can be added in the given time slot without violating the Unavailability constraint.

\* @return true if the constraint is satisfied

\*/

public boolean deltaValidateUnavailabilityConstraint(int day, int period, int course) {

return !unavailability.courseUnavailable[day][period][course];

}

/\*\*

\* Checks that a lecture can be added for a given course without violating the MaximimumLectures constraint

\* @return true if the constraint is not violated by the current schedule

\*/

public boolean deltaValidateMaximumScheduleCountConstraint(int course) {

return courseAssignmentCount[course] < courses.numberOfLecturesForCourse[course];

}

/\*\*

\* Checks that all constraints are satisfied if the given course is assigned to the given time slot.

\* @return true if the assignment does not violate constraint, false otherwise.

\*/

public boolean deltaValidateAllConstraints(int day, int period, int course) {

boolean allConstraintsSatisfied;

allConstraintsSatisfied = deltaValidateMaximumScheduleCountConstraint(course);

allConstraintsSatisfied = allConstraintsSatisfied && deltaValidateUnavailabilityConstraint(day, period, course);

allConstraintsSatisfied = allConstraintsSatisfied && deltaValidateSameLecturerConstraint(day, period, course);

allConstraintsSatisfied = allConstraintsSatisfied && deltaValidateSameCurriculumConstraint(day, period, course);

return allConstraintsSatisfied;

}

}

}

**DoNothingHeuristic.java**

/\*\*

\* Created by Martin on 05-04-2015.

\*/

public class DoNothingHeuristic extends Heuristic {

@Override

public Schedule search(Schedule schedule) {

return schedule;

}

}

**ExhaustiveTABU.java**

public class ExhaustiveTABU extends TABUHeuristic {

public boolean considerSwaps = true;

@Override

protected TABUOperationResult bestSolutionInNeighborhood() {

TABUOperationResult result = new TABUOperationResult();

result.scheduleValueAfterApplying = Integer.MAX\_VALUE;

for (int day = 0; day < basicInfo.days; day++) {

for (int period = 0; period < basicInfo.periodsPerDay; period++) {

for (int room = 0; room < basicInfo.rooms; room++) {

int assignedCourse = currentSchedule.assignments[day][period][room];

if (assignedCourse != EMPTY\_ROOM) {

int newValue = valueIfRemovingCourse(currentSchedule, currentScheduleValue, day, period, room);

if (newValue < result.scheduleValueAfterApplying) {

TABUOperation operation = new TABUOperation();

operation.day = day;

operation.period = period;

operation.room = room;

operation.type = OperationType.Remove;

if (!isTaboo(operation)) {

result.operation = operation;

result.scheduleValueAfterApplying = newValue;

}

}

// Consider swapping with all other courses

if (considerSwaps) {

for (int day2 = 0; day2 < basicInfo.days; day2++) {

for (int period2 = 0; period2 < basicInfo.periodsPerDay; period2++) {

for (int room2 = 0; room2 < basicInfo.rooms; room2++) {

int otherCourse = currentSchedule.assignments[day2][period2][room2];

if (otherCourse == EMPTY\_ROOM || otherCourse <= assignedCourse)

continue;

int newValueAfterSwap = valueIfSwappingCourses(currentSchedule, currentScheduleValue, day, period, room, day2, period2, room2);

if (newValueAfterSwap < result.scheduleValueAfterApplying) {

TABUOperation operation = new TABUOperation();

operation.day = day;

operation.period = period;

operation.room = room;

operation.otherDay = day2;

operation.otherPeriod = period2;

operation.otherRoom = room2;

operation.type = OperationType.Swap;

if (!isTaboo(operation)) {

result.operation = operation;

result.scheduleValueAfterApplying = newValueAfterSwap;

}

}

}

}

}

}

} else {

for (int course = 0; course < basicInfo.courses; course++) {

int newValue = valueIfAssigningCourse(currentSchedule, currentScheduleValue, day, period, room, course);

if (newValue < result.scheduleValueAfterApplying) {

TABUOperation operation = new TABUOperation();

operation.day = day;

operation.period = period;

operation.room = room;

operation.course = course;

operation.type = OperationType.Assign;

if (!isTaboo(operation)) {

result.operation = operation;

result.scheduleValueAfterApplying = newValue;

}

}

}

}

}

}

}

return result;

}

}

**StochasticHillClimber.java**

import java.io.FileWriter;

import java.io.IOException;

import java.util.Random;

import java.util.UUID;

import com.opencsv.CSVWriter;

/\*\*

\* Created by Burak on 08-04-2015.

\*/

public class StochasticHillClimber extends Heuristic {

protected int currentValue;

protected int previousValue = Integer.MAX\_VALUE;

private Random random = new XORShiftRandom();

public StochasticHillClimber() throws IOException {

super();

String uuid = UUID.randomUUID().toString();

f = new FileWriter(this.getClass()+uuid+"iterationValue.csv");

writer = new CSVWriter(f, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

}

@Override

public Schedule search(Schedule schedule) throws IOException {

startCountdown();

currentValue = evaluationFunction(schedule); // value of the current solution

deltaState.courseAssignmentCount = getCourseAssignmentCount(schedule);

String[] result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

int rooms = this.basicInfo.rooms;

int days = this.basicInfo.days;

int periods = this.basicInfo.periodsPerDay;

while (!timeoutReached()) {

this.iterationCount++; //Adds to the iteration count

// Find a candidate for changing

int room = random.nextInt(rooms);

int day = random.nextInt(days);

int period = random.nextInt(periods);

// If the room is empty, check the impact of adding a course

int currentlyAssignedCourse = schedule.assignments[day][period][room];

if (currentlyAssignedCourse == EMPTY\_ROOM) {

// Find a course to assign

int courseToAssign = random.nextInt(basicInfo.courses);

int valueIfThisCourseIsAssigned = valueIfAssigningCourse(schedule, currentValue, day, period, room, courseToAssign);

if (valueIfThisCourseIsAssigned < currentValue) {

assignCourse(schedule, day, period, room, courseToAssign);

currentValue = valueIfThisCourseIsAssigned;

}

} else {

// Maybe we should remove the assigned course?

int valueIfThisCourseIsRemoved = valueIfRemovingCourse(schedule, currentValue, day, period, room);

if (valueIfThisCourseIsRemoved < currentValue) {

removeCourse(schedule, day, period, room);

currentValue = valueIfThisCourseIsRemoved;

}

}

if((float)Math.abs(previousValue-currentValue)/currentValue >= 0.05 ) {

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

previousValue = currentValue;

}

}

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

writer.flush();

f.close();

return schedule;

}

}

**StochasticTabu.java**

import java.io.FileWriter;

import java.io.IOException;

import java.util.Random;

import java.util.UUID;

import java.util.Vector;

import javax.swing.text.html.HTMLDocument.HTMLReader.IsindexAction;

import com.opencsv.CSVWriter;

/\*\*

\* Created by Burak on 08-04-2015.

\*/

public class StochasticTABU extends Heuristic {

protected int previousBestValue = Integer.MAX\_VALUE;

private int tabooLength;

private int bestCourse1;

private int bestCourse2;

private Random random = new XORShiftRandom();

protected Vector<Integer> tabooList1; //The first taboolist - ONLY TABOOSEARCH

protected Vector<Integer> tabooList2; //The second taboolist - ONLY TABOOSEARCH

private Integer[] bestdayPeriodRoom1;

private Integer [] bestdayPeriodRoom2;

protected Vector<Integer[]> tabooListSlots1; //The first taboolist - ONLY TABOOSEARCH

protected Vector<Integer[]> tabooListSlots2; //The second taboolist - ONLY TABOOSEARCH

private static final int ASSIGNNO = -2;

private static final int REMOVENO = -3;

public StochasticTABU(int TabooLength) throws IOException

{

this.tabooLength = TabooLength;

//tabooList1 = new Vector<Integer>();

//tabooList2 = new Vector<Integer>();

tabooListSlots1 = new Vector<Integer[]>();

tabooListSlots2 = new Vector<Integer[]>();

String uuid = UUID.randomUUID().toString();

f = new FileWriter(this.getClass()+Integer.toString(tabooLength)+uuid+"iterationValue.csv");

writer = new CSVWriter(f, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

}

@Override

public Schedule search(Schedule currentSchedule) throws IOException {

startCountdown();

Schedule currentBestSchedule;

Schedule bestSchedule;

currentBestSchedule =new Schedule(this.basicInfo.days, this.basicInfo.periodsPerDay, this.basicInfo.rooms);

bestSchedule =new Schedule(this.basicInfo.days, this.basicInfo.periodsPerDay, this.basicInfo.rooms);

cloneArray(currentSchedule.assignments, currentBestSchedule.assignments);

int currentValue;

int currentBestValue ;

int bestValue = Integer.MAX\_VALUE;

currentValue = evaluationFunction(currentSchedule); // value of the current solution

String[] result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

deltaState.courseAssignmentCount = getCourseAssignmentCount(currentSchedule);

int rooms = this.basicInfo.rooms;

int days = this.basicInfo.days;

int periods = this.basicInfo.periodsPerDay;

System.out.println("Start");

while(timeoutReached() == false) {

currentBestValue = Integer.MAX\_VALUE;

this.iterationCount++; //Adds to the iteration count

for(int day=0;day<this.basicInfo.days;day++) { //run thorough all the days

for(int period = 0;period<this.basicInfo.periodsPerDay;period++) { //all the periods

for(int room = 0;room<this.basicInfo.rooms;room++){ //all the rooms

int room2 = random.nextInt(rooms);

int day2 = random.nextInt(days);

int period2 = random.nextInt(periods);

while((day==day2)&&(period==period2)&&(room==room2) ) {

room2 = random.nextInt(rooms);

day2 = random.nextInt(days);

period2 = random.nextInt(periods);

}

int valueIfThisCourseIsAssigned = Integer.MAX\_VALUE;

int valueIfThisCourseIsRemoved = Integer.MAX\_VALUE;

int valueIfThisCoursesAreSwapped = Integer.MAX\_VALUE;

int courseId = random.nextInt(this.basicInfo.courses);

if(currentSchedule.assignments[day][period][room] != StochasticHillClimber.EMPTY\_ROOM) {

valueIfThisCourseIsRemoved = valueIfRemovingCourse(currentSchedule, currentValue, day,period, room ); //calculate the value if we remove the course given timeslot

}

else {

valueIfThisCourseIsAssigned = valueIfAssigningCourse(currentSchedule, currentValue, day,period, room , courseId); //calculate the value if we swap the courses given timeslots

}

valueIfThisCoursesAreSwapped = valueIfSwappingCourses(currentSchedule, currentValue, day, period, room, day2, period2, room2);//calculate the value if we add the course given timeslot

//System.err.println("valueIfThisCourseIsRemoved = " + valueIfThisCourseIsRemoved + " valueIfThisCourseIsAssigned ="+ valueIfThisCourseIsAssigned+ " valueIfThisCoursesAreSwapped =" + valueIfThisCoursesAreSwapped);

/\*if(currentValue<0) {

System.err.println("currentValue = "+currentValue );

System.exit(0);

}

System.err.println("valueIfThisCoursesAreSwapped = "+valueIfThisCoursesAreSwapped );

System.err.println("valueIfThisCourseIsAssigned = "+valueIfThisCourseIsAssigned );

System.err.println("valueIfThisCourseIsRemoved = "+valueIfThisCourseIsRemoved );

\*/

Type change;

//we have the new values now we need to choose which action would be the best according to new values (find the best neighboor)

if(valueIfThisCourseIsRemoved<=valueIfThisCourseIsAssigned){

if(valueIfThisCourseIsRemoved<=valueIfThisCoursesAreSwapped) {

if(valueIfThisCourseIsRemoved != Integer.MAX\_VALUE)

change = Type.REMOVE;//if removing the course gives the best value then choose remove

else

change = Type.NOTHING;//that means we have Max\_int value so we do nothing in this iteration

}

else {

change = Type.SWAP;//choose swap if the swapping gives the best value

}

}

else {

if(valueIfThisCourseIsAssigned<valueIfThisCoursesAreSwapped)

change = Type.ASSIGN; //choose assign if the assigning gives the best value

else

change = Type.SWAP;//choose swap if the swapping gives the best value

}

//System.err.println("" + currentBestValue);

switch (change) {

case REMOVE:{

bestdayPeriodRoom1 = new Integer[3];

bestdayPeriodRoom1[0] = day;

bestdayPeriodRoom1[1] = period;

bestdayPeriodRoom1[2] = room;

bestdayPeriodRoom2 = new Integer[3];

bestdayPeriodRoom2[0] = REMOVENO;

bestdayPeriodRoom2[1] = REMOVENO;

bestdayPeriodRoom2[2] = REMOVENO;

if(valueIfThisCourseIsRemoved < currentBestValue && !IsTaboo(bestdayPeriodRoom1,bestdayPeriodRoom2)) {

currentBestValue = valueIfThisCourseIsRemoved;

int courseName = currentSchedule.assignments[day][period][room];//Remembers the course for change back

removeCourse(currentSchedule, day, period, room);

cloneArray(currentSchedule.assignments, currentBestSchedule.assignments);

assignCourse(currentSchedule, day, period, room, courseName);

}

//swap back changes in current schedule

break;

}

case ASSIGN: {

bestdayPeriodRoom1 = new Integer[3];

bestdayPeriodRoom1[0] = day;

bestdayPeriodRoom1[1] = period;

bestdayPeriodRoom1[2] = room;

bestdayPeriodRoom2 = new Integer[3];

bestdayPeriodRoom2[0] = ASSIGNNO;

bestdayPeriodRoom2[1] = ASSIGNNO;

bestdayPeriodRoom2[2] = ASSIGNNO;

if(valueIfThisCourseIsAssigned < currentBestValue && !IsTaboo(bestdayPeriodRoom1,bestdayPeriodRoom2)) {

assignCourse(currentSchedule, day, period, room, courseId);

currentBestValue = valueIfThisCourseIsAssigned;

cloneArray(currentSchedule.assignments, currentBestSchedule.assignments);

//swap back changes in current schedule

removeCourse(currentSchedule, day, period, room);

}

break;

}

case SWAP: {

bestdayPeriodRoom1 = new Integer[3];

bestdayPeriodRoom1[0] = day;

bestdayPeriodRoom1[1] = period;

bestdayPeriodRoom1[2] = room;

bestdayPeriodRoom2 = new Integer[3];

bestdayPeriodRoom2[0] = day2;

bestdayPeriodRoom2[1] = period2;

bestdayPeriodRoom2[2] = room2;

if(valueIfThisCoursesAreSwapped < currentBestValue && !IsTaboo(bestdayPeriodRoom1,bestdayPeriodRoom2)) {

bestCourse1 = currentSchedule.assignments[day][period][room];//Remembers the course for to assign

bestCourse2 = currentSchedule.assignments[day2][period2][room2]; //Remembers the course for to assign

removeCourse(currentSchedule, day2, period2, room2);

removeCourse(currentSchedule, day, period, room);

assignCourse(currentSchedule, day2, period2, room2, bestCourse1);

assignCourse(currentSchedule, day, period, room, bestCourse2);

currentBestValue = valueIfThisCoursesAreSwapped;

cloneArray(currentSchedule.assignments, currentBestSchedule.assignments);

//swap back changes in current schedule

removeCourse(currentSchedule, day2, period2, room2);

removeCourse(currentSchedule, day, period, room);

assignCourse(currentSchedule, day, period, room, bestCourse1);

assignCourse(currentSchedule, day2, period2, room2, bestCourse2);

}

break;

}

default:

break;

}

}

}

}

//save the current best schedule we are going to start searching there next iteration

cloneArray( currentBestSchedule.assignments,currentSchedule.assignments);

if(currentBestValue!=Integer.MAX\_VALUE)

currentValue = currentBestValue;

AddTaboo(bestdayPeriodRoom1, bestdayPeriodRoom2);//we add the course in the tabo list with the REMOVENO so when we check the taboo list we will know its assigned

if( currentBestValue<bestValue) {

bestValue = currentBestValue;

cloneArray( currentBestSchedule.assignments,bestSchedule.assignments);

}

if((float)Math.abs(previousBestValue-bestValue)/bestValue >= 0.05 ) {

result = new String[] { "" + iterationCount, bestValue + "" };

writer.writeNext(result);

previousBestValue = bestValue;

}

}

result = new String[] { "" + iterationCount, bestValue + "" };

writer.writeNext(result);

writer.flush();

f.close();

//System.out.println("Tabu Found A Solution!");

//System.out.println("Value = "+evaluationFunction(currentBestSchedule));

cloneArray( bestSchedule.assignments,currentBestSchedule.assignments);

return currentBestSchedule;

}

/\*\*

\* Adds this swap to the taboolist - ONLY TABOOSEARCH

\* @param course1

\* @param course2

\*/

public void AddTaboo(int course1, int course2)

{

//If the list is full, the first added is now removed

if(this.tabooList1.size() == this.tabooLength)

{

this.tabooList1.remove(0);

this.tabooList2.remove(0);

}

this.tabooList1.add(course1);

this.tabooList2.add(course2);

}

/\*\*

\* Finds out if the swap is tabooed - ONLY TABOO SEARCH

\* @param course1

\* @param course2

\* @return

\*/

public boolean IsTaboo(int course1, int course2)

{

for (int i = 0 ; i < this.tabooList1.size(); i++)

{

if(course2==ASSIGNNO) { //that means we need to check TABU list for assign

if(tabooList1.elementAt(i) == course1)

{

if(tabooList2.elementAt(i) == REMOVENO) //that means if we remove the course before dont assign that course again it is TABU

{

return true;

}

}

}

else if (course2 == REMOVENO) {//check TABU list for remove

if(tabooList1.elementAt(i) == course1)

{

if(tabooList2.elementAt(i) == ASSIGNNO) //that means if we assign the course before dont remove that course again it is TABU

{

return true;

}

}

}

else { //finaly if course2 is not ASSIGNNO or REMOVENO then check TABU list for Swap

if(tabooList1.elementAt(i) == course1)

{

if(tabooList2.elementAt(i) == course2)

{

return true;

}

}

if(tabooList1.elementAt(i) == course2)

{

if(tabooList2.elementAt(i) == course1)

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Adds this swap to the taboolist - ONLY TABOOSEARCH

\* @param dayPeriodRoom1

\* @param dayPeriodRoom2

\*/

public void AddTaboo(Integer[] dayPeriodRoom1,Integer[] dayPeriodRoom2)

{

//If the list is full, the first added is now removed

if(this.tabooListSlots1.size() == this.tabooLength)

{

this.tabooListSlots1.remove(0);

this.tabooListSlots1.remove(0);

}

this.tabooListSlots1.add(dayPeriodRoom1);

this.tabooListSlots2.add(dayPeriodRoom2);

}

/\*\*

\* Finds out if the swap is tabooed - ONLY TABOO SEARCH

\*/

public boolean IsTaboo(Integer[] dayPeriodRoom1,Integer[] dayPeriodRoom2)

{

if(dayPeriodRoom2[0] == REMOVENO) {

for (int i = 0 ; i < this.tabooListSlots1.size(); i++)

{

if(tabooListSlots1.elementAt(i)[0] == dayPeriodRoom1[0] && tabooListSlots1.elementAt(i)[1] == dayPeriodRoom1[1] && tabooListSlots1.elementAt(i)[2] == dayPeriodRoom1[2] )

{

if(tabooListSlots2.elementAt(i)[0] == ASSIGNNO)

{

return true;

}

}

}

}

else if (dayPeriodRoom2[0] == ASSIGNNO) {

for (int i = 0 ; i < this.tabooListSlots1.size(); i++)

{

//System.err.println(" size = " + tabooListSlots1.size() + " index = " +i);

if(tabooListSlots1.elementAt(i)[0] == dayPeriodRoom1[0] && tabooListSlots1.elementAt(i)[1] == dayPeriodRoom1[1] && tabooListSlots1.elementAt(i)[2] == dayPeriodRoom1[2] )

{

if(tabooListSlots2.elementAt(i)[0] == REMOVENO)

{

return true;

}

}

}

}

else {

for (int i = 0 ; i < this.tabooListSlots1.size(); i++)

{

if(tabooListSlots1.elementAt(i)[0] == dayPeriodRoom1[0] && tabooListSlots1.elementAt(i)[1] == dayPeriodRoom1[1] && tabooListSlots1.elementAt(i)[2] == dayPeriodRoom1[2] )

{

if(tabooListSlots2.elementAt(i)[0] == dayPeriodRoom2[0] && tabooListSlots2.elementAt(i)[1] == dayPeriodRoom2[1] && tabooListSlots2.elementAt(i)[2] == dayPeriodRoom2[2])

{

return true;

}

}

if(tabooListSlots1.elementAt(i)[0] == dayPeriodRoom2[0] && tabooListSlots1.elementAt(i)[1] == dayPeriodRoom2[1] && tabooListSlots1.elementAt(i)[2] == dayPeriodRoom2[2] )

{

if(tabooListSlots2.elementAt(i)[0] == dayPeriodRoom1[0] && tabooListSlots2.elementAt(i)[1] == dayPeriodRoom1[1] && tabooListSlots2.elementAt(i)[2] == dayPeriodRoom1[2])

{

return true;

}

}

}

}

return false;

}

}

**TABU.java**

import java.io.FileWriter;

import java.io.IOException;

import java.lang.reflect.Array;

import java.util.UUID;

import java.util.Vector;

import com.opencsv.CSVWriter;

/\*\*

\* Created by Burak on 08-04-2015.

\*/

public class TABU extends Heuristic {

protected Schedule schedule; //current schedule

protected Schedule currentSchedule; //the copy of the current schedule where changes are made, that are not certain to be saved

protected int currentValue;

protected int previousValue = Integer.MAX\_VALUE;

private int tabooLength;

private Integer[] bestdayPeriodRoom1;

private Integer [] bestdayPeriodRoom2;

protected Vector<Integer[]> tabooList1; //The first taboolist - ONLY TABOOSEARCH

protected Vector<Integer[]> tabooList2; //The second taboolist - ONLY TABOOSEARCH

public TABU(int TabooLength) throws IOException

{

super();

this.tabooLength = TabooLength;

tabooList1 = new Vector<Integer[]>();

tabooList2 = new Vector<Integer[]>();

String uuid = UUID.randomUUID().toString();

f = new FileWriter(this.getClass()+Integer.toString(tabooLength)+uuid+"iterationValue.csv");

writer = new CSVWriter(f, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

}

@Override

public Schedule search(Schedule schedule) throws IOException {

startCountdown();

currentValue = evaluationFunction(schedule); // value of the current solution

deltaState.courseAssignmentCount = getCourseAssignmentCount(schedule);

String[] result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

System.out.println("Start");

while(timeoutReached() == false) {

this.iterationCount++; //Adds to the iteration count

for(int day=0;day<this.basicInfo.days;day++) { //run thorough all the days

for(int period = 0;period<this.basicInfo.periodsPerDay;period++) { //all the periods

for(int room = 0;room<this.basicInfo.rooms;room++){ //all the rooms

for(int day2=0;day2<this.basicInfo.days;day2++) { //all the days can be swap to

for(int period2 = 0;period2<this.basicInfo.periodsPerDay;period2++) { //all the periods can be swap to

for(int room2 = 0;room2<this.basicInfo.rooms;room2++){ //all the rooms can be swap to

//int valueIfThisCourseIsAssigned = Integer.MAX\_VALUE;

//int valueIfThisCourseIsRemoved = Integer.MAX\_VALUE;

int valueIfThisCoursesAreSwapped = Integer.MAX\_VALUE;

//TODO:also check the values if with emoving and adding methods

valueIfThisCoursesAreSwapped = valueIfSwappingCourses(schedule, currentValue, day, period,room, day2, period2, room2);

if(currentValue>valueIfThisCoursesAreSwapped && IsTaboo(day,period,room, day2,period2,room2) == false) {

currentValue = valueIfThisCoursesAreSwapped;

bestdayPeriodRoom1 = new Integer[3];

bestdayPeriodRoom1[0] = day;

bestdayPeriodRoom1[1] = period;

bestdayPeriodRoom1[2] = room;

bestdayPeriodRoom2 = new Integer[3];

bestdayPeriodRoom2[0] = day2;

bestdayPeriodRoom2[1] = period2;

bestdayPeriodRoom2[2] = room2;

int bestCourse1 = schedule.assignments[day][period][room];//Remembers the course for the assign

int bestCourse2 = schedule.assignments[day2][period2][room2]; //Remembers the course for the assign

removeCourse(schedule, day2, period2, room2);

removeCourse(schedule, day, period, room);

assignCourse(schedule, day2, period2, room2, bestCourse1);

assignCourse(schedule, day, period, room, bestCourse2);

}

}

}

}

}

}

}

AddTaboo(bestdayPeriodRoom1, bestdayPeriodRoom2); //Makes the swap back taboo.

if((float)Math.abs(previousValue-currentValue)/currentValue >= 0.05 ) {

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

previousValue = currentValue;

}

}

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

writer.flush();

f.close();

System.out.println("Tabu Found A Solution!");

System.out.println("Value = "+evaluationFunction(schedule));

return schedule;

}

/\*\*

\* Adds this swap to the taboolist - ONLY TABOOSEARCH

\* @param dayPeriodRoom1

\* @param dayPeriodRoom2

\*/

public void AddTaboo(Integer[] dayPeriodRoom1,Integer[] dayPeriodRoom2)

{

//If the list is full, the first added is now removed

if(this.tabooList1.size() == this.tabooLength)

{

this.tabooList1.remove(0);

this.tabooList2.remove(0);

}

this.tabooList1.add(dayPeriodRoom1);

this.tabooList2.add(dayPeriodRoom2);

}

/\*\*

\* Finds out if the swap is tabooed - ONLY TABOO SEARCH

\* @param course1

\* @param course2

\* @return

\*/

public boolean IsTaboo(int day1,int period1, int room1,int day2,int period2,int room2)

{

for (int i = 0 ; i < this.tabooList1.size(); i++)

{

if(tabooList1.elementAt(i)[0] == day1 && tabooList1.elementAt(i)[1] == period1 && tabooList1.elementAt(i)[2] == room1 )

{

if(tabooList2.elementAt(i)[0] == day2 && tabooList2.elementAt(i)[1] == period2 && tabooList2.elementAt(i)[2] == room2)

{

return true;

}

}

if(tabooList1.elementAt(i)[0] == day2 && tabooList1.elementAt(i)[1] == period2 && tabooList1.elementAt(i)[2] == room2 )

{

if(tabooList2.elementAt(i)[0] == day1 && tabooList2.elementAt(i)[1] == period1 && tabooList2.elementAt(i)[2] == room1)

{

return true;

}

}

}

return false;

}

}

**TABUHeuristic.java**

import sun.awt.image.ImageWatched;

import java.io.IOException;

import java.util.LinkedList;

public abstract class TABUHeuristic extends Heuristic {

protected Schedule currentSchedule, bestSchedule;

protected int currentScheduleValue, bestScheduleValue;

public enum OperationType {

Assign,

Remove,

Swap

}

public int tabooListLength = 20;

public class TABUOperation {

public int day = -1;

public int period = -1;

public int room = -1;

public int course = -1;

public int otherDay = -1;

public int otherPeriod = -1;

public int otherRoom = -1;

public int otherCourse = -1;

public OperationType type;

}

public class TABUOperationResult {

public TABUOperation operation;

public int scheduleValueAfterApplying;

}

private void applyOperation(Schedule schedule, TABUOperation operation) {

if (operation == null)

return;

switch (operation.type) {

case Assign:

assignCourse(schedule, operation.day, operation.period, operation.room, operation.course);

break;

case Remove:

removeCourse(schedule, operation.day, operation.period, operation.room);

break;

case Swap:

removeCourse(schedule, operation.day, operation.period, operation.room);

removeCourse(schedule, operation.otherDay, operation.otherPeriod, operation.otherRoom);

assignCourse(schedule, operation.day, operation.period, operation.room, operation.otherCourse);

assignCourse(schedule, operation.otherDay, operation.otherPeriod, operation.otherRoom, operation.course);

break;

}

}

@Override

public Schedule search(Schedule schedule) throws IOException {

currentSchedule = schedule;

currentScheduleValue = evaluationFunction(currentSchedule);

bestSchedule = new Schedule(basicInfo.days, basicInfo.periodsPerDay, basicInfo.rooms);

cloneArray(schedule.assignments, bestSchedule.assignments);

bestScheduleValue = currentScheduleValue;

startCountdown();

while(!timeoutReached()) {

iterationCount++;

// Determine the operation needed to get the best solution in neighborhood

TABUOperationResult operation = bestSolutionInNeighborhood();

// Apply that operation to the currentSchedule

applyOperation(currentSchedule, operation.operation);

currentScheduleValue = operation.scheduleValueAfterApplying;

// If we just encountered a better solution than earlier, store it

if (currentScheduleValue < bestScheduleValue) {

cloneArray(schedule.assignments, bestSchedule.assignments);

bestScheduleValue = currentScheduleValue;

}

}

return bestSchedule;

}

protected LinkedList<TABUOperation> tabooList = new LinkedList<TABUOperation>();

private void addTaboo(TABUOperation operation) {

tabooList.addLast(operation);

while (tabooList.size() > tabooListLength)

tabooList.removeFirst();

}

protected boolean isTaboo(TABUOperation operation) {

switch (operation.type) {

case Assign:

case Swap:

for (TABUOperation taboo : tabooList) {

if (taboo.course == operation.course || taboo.course == operation.otherCourse || taboo.otherCourse == operation.course || taboo.otherCourse == operation.otherCourse)

return true;

}

return false;

case Remove:

default:

for (TABUOperation taboo : tabooList) {

if (taboo.day == operation.day && taboo.period == operation.period && taboo.room == operation.room)

return true;

}

return false;

}

}

protected abstract TABUOperationResult bestSolutionInNeighborhood();

}

**HillClimber.java**

import java.util.HashMap;

import java.util.Map;

import java.util.Random;

/\*\*

\* Created by Burak on 08-04-2015.

\*/

public class HillClimber extends Heuristic {

protected Schedule schedule; //current schedule

//protected Schedule currentSchedule; //the copy of the current schedule where changes are made, that are not certain to be saved

protected int IterationCount = 0;

protected int currentValue;

private Random Rand = new XORShiftRandom();

private Map<Integer, Integer> unScheduledCourses;

@Override

public Schedule search(Schedule schedule) {

startCountdown();

currentValue = evaluationFunction(schedule); // value of the current solution

deltaState.courseAssignmentCount = getCourseAssignmentCount(schedule);

unScheduledCourses = new HashMap<Integer, Integer>();

for (int courseNo = 0; courseNo < this.basicInfo.courses; courseNo++) { //checked the unscheduled course and the number of unscheduled lectures

int numberOfLecturesUnAssigned = this.deltaState.courseAssignmentCount[courseNo] - this.courses.numberOfLecturesForCourse[courseNo];

if (numberOfLecturesUnAssigned != 0)

unScheduledCourses.put(courseNo, Math.abs(numberOfLecturesUnAssigned));

}

boolean done = false;

System.out.println("Start");

while (timeoutReached() == false) { //TODO: we need to deal with the d

done = true; //Runs while there are still changes being made

//System.out.println("Iteration Count = " + IterationCount);

this.IterationCount++; //Adds to the iteration count

if (this.IterationCount % 1000 == 0)

System.err.println("Iteration Count = " + this.IterationCount);

int currentBestValue = currentValue;

for (int day = 0; day < this.basicInfo.days; day++) { //run thorough all the days

for (int period = 0; period < this.basicInfo.periodsPerDay; period++) { //all the periods

for (int room = 0; room < this.basicInfo.rooms; room++) { //all the rooms

// System.out.println("day = "+ day +" period = " + period + " room = " + room + " course = " + schedule.assignments[day][period][room]);

for (Map.Entry<Integer, Integer> entry : unScheduledCourses.entrySet()) {

int unScheduledCourseNo = entry.getKey();

int unScheduledCourseValue = entry.getValue();

if (schedule.assignments[day][period][room] == Heuristic.EMPTY\_ROOM) {// room is empty

int valueIfThisCourseIsAssigned = valueIfAssigningCourse(schedule, currentValue, day, period, room, unScheduledCourseNo);

if (valueIfThisCourseIsAssigned <= currentValue) {

assignCourse(schedule, day, period, room, unScheduledCourseNo);

unScheduledCourseValue--;

if (unScheduledCourseValue == 0) {

unScheduledCourses.remove(unScheduledCourseNo);

}

}

} else { //room is not epmty first we need to check the

}

// remove the course in current time slot and then add the unscheduled course

removeCourse(schedule, day, period, room);

assignCourse(schedule, day, period, room, unScheduledCourseNo);

}

int valueIfThisCourseIsAssigned, valueIfThisCourseIsRemoved, valueIfThisCoursesAreSwapped;

int day2, period2, room2;

day2 = Rand.nextInt(this.basicInfo.days);

room2 = Rand.nextInt(this.basicInfo.rooms);

period2 = Rand.nextInt(this.basicInfo.periodsPerDay);

valueIfThisCourseIsRemoved = valueIfRemovingCourse(schedule, currentValue, day, period, room);

int courseId = Rand.nextInt(this.basicInfo.courses);

valueIfThisCourseIsAssigned = valueIfAssigningCourse(schedule, currentValue, day, period, room, courseId);

valueIfThisCoursesAreSwapped = valueIfSwappingCourses(schedule, currentValue, day, period, room, day2, period2, room2);

Type change;

if (valueIfThisCourseIsRemoved <= valueIfThisCourseIsAssigned) {

if (valueIfThisCourseIsRemoved <= valueIfThisCoursesAreSwapped) {

if (valueIfThisCourseIsRemoved != Integer.MAX\_VALUE)

change = Type.REMOVE;

else

change = Type.NOTHING;

} else {

change = Type.SWAP;

}

} else {

if (valueIfThisCourseIsAssigned < valueIfThisCoursesAreSwapped)

change = Type.ASSIGN;

else

change = Type.SWAP;

}

int bestCourse1;

int bestCourse2;

int deltaval;

switch (change) {

case REMOVE: {

deltaval = valueIfThisCourseIsRemoved - currentValue;

if (deltaval < 0) {

currentValue += deltaval;

removeCourse(schedule, day, period, room);

}

break;

}

case ASSIGN: {

deltaval = valueIfThisCourseIsAssigned - currentValue;

if (deltaval < 0) {

currentValue += deltaval;

assignCourse(schedule, day, period, room, courseId);

}

break;

}

case SWAP: {

deltaval = valueIfThisCoursesAreSwapped - currentValue;

if (deltaval < 0) {

currentValue = valueIfThisCoursesAreSwapped;

bestCourse1 = schedule.assignments[day][period][room];//Remembers the person for the taboolist

bestCourse2 = schedule.assignments[day2][period2][room2]; //Remembers the person for the taboolist

removeCourse(schedule, day2, period2, room2);

removeCourse(schedule, day, period, room);

assignCourse(schedule, day2, period2, room2, bestCourse1);

assignCourse(schedule, day, period, room, bestCourse2);

}

break;

}

default:

break;

}

}

}

}

if (currentBestValue < currentValue) {

done = false;

currentValue = currentBestValue;

}

}

return schedule;

}

}

**SimulatedAnnealing.java**

import java.io.FileWriter;

import java.io.IOException;

import java.util.Random;

import java.util.UUID;

import com.opencsv.CSVWriter;

public class SimulatedAnnealing extends Heuristic{

protected Schedule schedule; //current schedule

protected Schedule currentSchedule; //the copy of the current schedule where changes are made, that are not certain to be saved

protected int currentValue;

protected int previousValue = Integer.MAX\_VALUE;

private double temperature;

private double tempchange;

private int deltaval;

private int day1;

private int period1;

private int room1;

private int day2;

private int period2;

private int room2;

private int bestCourse1;

private int bestCourse2;

private Random Rand = new XORShiftRandom();

public SimulatedAnnealing(double temperature,double tempchange) throws IOException {

super();

this.temperature = temperature;

this.tempchange = tempchange;

// write iteration value to a CSV file

String uuid = UUID.randomUUID().toString();

f = new FileWriter(this.getClass()+Float.toString((float) temperature)+Float.toString((float) tempchange)+uuid+"iterationValue.csv");

writer = new CSVWriter(f, ',', CSVWriter.NO\_QUOTE\_CHARACTER);

}

@Override

public Schedule search(Schedule schedule) throws IOException {

System.out.println("Start");

startCountdown();

currentValue = evaluationFunction(schedule); // value of the current solution

deltaState.courseAssignmentCount = getCourseAssignmentCount(schedule);

String[] result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

while(!timeoutReached()) {

this.iterationCount++; //Adds to the iteration count

if(iterationCount % 100000 == 0)

System.out.println("Iteation count = " + iterationCount);

day1 = day2 = period1 = period2 = room1 = room2 = 0;

boolean hardConstraintViolation = true;

int valueIfThisCourseIsAssigned = Integer.MAX\_VALUE;

int valueIfThisCourseIsRemoved = Integer.MAX\_VALUE;

int valueIfThisCoursesAreSwapped = Integer.MAX\_VALUE;

int courseId = -1;

while(((day1==day2)&&(period1==period2)&&(room1==room2) || hardConstraintViolation) ) {

day1 = Rand.nextInt(this.basicInfo.days);

day2 = Rand.nextInt(this.basicInfo.days);

period1 = Rand.nextInt(this.basicInfo.periodsPerDay);

period2 = Rand.nextInt(this.basicInfo.periodsPerDay);

room1 = Rand.nextInt(this.basicInfo.rooms);

room2 = Rand.nextInt(this.basicInfo.rooms);

valueIfThisCoursesAreSwapped = valueIfSwappingCourses(schedule, currentValue, day1, period1,room1,day2,period2,room2);

valueIfThisCourseIsRemoved = valueIfRemovingCourse(schedule, currentValue, day1, period1, room1);

courseId = Rand.nextInt(this.basicInfo.courses);

valueIfThisCourseIsAssigned = valueIfAssigningCourse(schedule, currentValue, day1, period1, room1, courseId);

if(!( valueIfThisCourseIsRemoved == Integer.MAX\_VALUE && valueIfThisCourseIsAssigned == Integer.MAX\_VALUE))

hardConstraintViolation = false;

}

Type change;

if(valueIfThisCourseIsRemoved<=valueIfThisCourseIsAssigned){

if(valueIfThisCourseIsRemoved<=valueIfThisCoursesAreSwapped) {

if(valueIfThisCourseIsRemoved != Integer.MAX\_VALUE)

change = Type.REMOVE;

else

change = Type.NOTHING;

}

else {

change = Type.SWAP;

}

}

else {

if(valueIfThisCourseIsAssigned<valueIfThisCoursesAreSwapped)

change = Type.ASSIGN;

else

change = Type.SWAP;

}

switch (change) {

case REMOVE:{

deltaval = valueIfThisCourseIsRemoved - currentValue;

if(deltaval < 0) {

currentValue +=deltaval;

removeCourse(schedule, day1, period1, room1);

}

else if(GetProbability() > Rand.nextDouble() && deltaval!=0) {

currentValue +=deltaval;

removeCourse(schedule, day1, period1, room1);

}

break;

}

case ASSIGN: {

deltaval = valueIfThisCourseIsAssigned - currentValue;

if(deltaval < 0) {

currentValue +=deltaval;

assignCourse(schedule, day1, period1, room1, courseId);

}

else if(GetProbability() > Rand.nextDouble() && deltaval!=0) {

currentValue +=deltaval;

assignCourse(schedule, day1, period1, room1, courseId);

}

break;

}

case SWAP: {

deltaval = valueIfThisCoursesAreSwapped - currentValue;

if(deltaval < 0) {

currentValue = valueIfThisCoursesAreSwapped;

bestCourse1 = schedule.assignments[day1][period1][room1];//Remembers the person for the taboolist

bestCourse2 = schedule.assignments[day2][period2][room2]; //Remembers the person for the taboolist

removeCourse(schedule, day2, period2, room2);

removeCourse(schedule, day1, period1, room1);

assignCourse(schedule, day2, period2, room2, bestCourse1);

assignCourse(schedule, day1, period1, room1, bestCourse2);

}

else if(GetProbability() > Rand.nextDouble() && deltaval!=0 ) {

currentValue = valueIfThisCoursesAreSwapped;

bestCourse1 = schedule.assignments[day1][period1][room1];//Remembers the person for the taboolist

bestCourse2 = schedule.assignments[day2][period2][room2]; //Remembers the person for the taboolist

removeCourse(schedule, day2, period2, room2);

removeCourse(schedule, day1, period1, room1);

assignCourse(schedule, day2, period2, room2, bestCourse1);

assignCourse(schedule, day1, period1, room1, bestCourse2);

}

break;

}

default:

break;

}

// TODO write results to a CSV file

temperature= temperature\*tempchange; //Reduces the temperature

if((float)Math.abs(previousValue-currentValue)/currentValue >= 0.05 ) {

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

previousValue = currentValue;

}

}

result = new String[] { "" + iterationCount, currentValue + "" };

writer.writeNext(result);

writer.flush();

f.close();

return schedule;

}

private double GetProbability()

{

return Math.exp(-deltaval/temperature);

}

}