# Choosing a Suitable 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.

# 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.