The Design of a Course-Timetabling System Using Graph-Coloring and Artificial Intelligence

Jordan P. Rickman
Rollins College, jrickman@rollins.edu

Follow this and additional works at: http://scholarship.rollins.edu/honors

Part of the Other Applied Mathematics Commons

Recommended Citation

This is brought to you for free and open access by Rollins Scholarship Online. It has been accepted for inclusion in Honors Program Theses by an authorized administrator of Rollins Scholarship Online. For more information, please contact wzhang@rollins.edu.
THE DESIGN OF A COURSE-TIMETABLING SYSTEM USING
GRAPH-COLORING AND ARTIFICIAL INTELLIGENCE

Jordan Rickman
Jay Yellen, Ph.D.

Senior Thesis for Artium Baccalaureus Honoris
Department of Mathematics and Computer Science, Rollins College
1000 Holt Ave, Winter Park, FL 32789
Abstract

We describe our research on the course-timetabling problem, which we model using graph-coloring. We focus on the development of a high-level algorithm that approaches the problem via state-space search using a priority queue. The speed and effectiveness of this algorithm varies based on the properties of two key components: the priority function and the branching procedure. We detail the challenges of developing each component, our current approaches to these challenges, and possible directions of future research.

For testing our algorithm, it is desirable to have a large set of characteristic course-timetabling problems to work with. To this end, we present our implementation of a random problem generator. It begins with existing problems as seeds and uses techniques adapted from genetic learning to generate similar problems. We also describe how the resulting sets of random problems could be used to improve our priority queue algorithm via machine learning.

Lastly, we describe the Java implementation of our algorithm, including the high-level structure of the code and solutions to specific problems encountered in the implementation process.

I. The Course-Timetabling Problem

Timetabling problems are operations research problems in which events are assigned timeslots under various constraints. Study of timetabling problems has included a wide range of optimization techniques, including integer programming (e.g. Boland, Hughes, et al. 2008), search methods (e.g. Gaspero and Schaerf 2000; Burke, Kendall, Soubeiga 2003), and machine learning (e.g. Wilke, Gröbner, Oster 2002; Frausto-Solis et al. 2006).

Our research considers timetabling problems of the same form as those considered in Wehrer and Yellen (2013). The events are course offerings such as those at a university and the timeslots are collections of days and times during the week. Constraints include hard and soft constraints regarding avoidance of scheduling conflicts, suitability of rooms and timeslots, and creation of compact schedules. Constraints are captured by means of a penalty function, or total penalty. Violation of soft constraints increases the total penalty, and an optimal solution is a timetable that minimizes the total penalty. Hard constraints can also be captured in the penalty function - they are given prohibitively high penalties, such that it will be less costly to violate soft constraints many times than to violate a hard constraint.

Constraints and Penalties

The most central constraint in our work is the avoidance of scheduling conflicts. Pairs of courses are “in conflict” when there is a particular reason these courses should not take place at the same time. Sources of conflict between courses include being taught by the same instructor, needing the same special classroom or equipment, or an expectation that some students will be taking both courses. Since Rollins schedules courses before
students sign up for them, the number of students taking both courses is not known, but can usually be estimated based on past experience.

In our work, we categorize conflicts into three categories of severity: light, medium, and heavy. Light and medium conflicts represent soft constraints, usually as a result of a few students tending to take both courses. Heavy conflicts are hard constraints with a prohibitive penalty. Sources of heavy conflict include mutual usage of an exclusive resource, mutual instructors, or an expectation that a significant number of students will need to take both courses concurrently. If two courses in conflict are assigned to overlapping timeslots, the conflict is said to be incurred, and an associated conflict penalty is added to the total penalty. Light conflicts incur a penalty of 1, medium conflicts incur a penalty of 20, and heavy conflicts incur a penalty of 400. These penalty weights can be chosen arbitrarily; our choices are taken from Wehrer and Yellen (2013).

The second central constraint represents a softer objective for the problem: the creation of compact schedules for students and instructors. Compact schedules are those that avoid large gaps between classes on the same day, and are usually desirable for both students and instructors. For instance, most professors do not want to teach a 3 PM class on the same day as an 8 AM class, and students would not want to take this combination of classes either. Proximity penalties also vary in their severity. We assign each pair of courses a proximity severity proportional to the number of students expected to take both courses and the number of instructors in common, where instructors are given a higher relative weight (where we use “proximity severity,” Wehrer and Yellen (2013) use the term “overlap factor”). The compactness of instructors’ schedules is given higher relative weight, by a factor of fifteen. The proximity penalty incurred for the pair is the product of the proximity severity and the gap factor, a function of the two timeslots assigned to the courses. The gap factor increases with the size of the gap between two timeslots on common days, though it is zero when this gap is small or there are no days in common. Whenever a nonzero proximity penalty is incurred, we call this occurrence a proximity violation.

Further constraints include the suitability of courses for timeslots, and room availability. Here assignment to an unsuitable timeslot - for instance, of a lab course to a non-laboratory timeslot - incurs a prohibitive penalty of 10,000. Significantly smaller penalties may be incurred if the timeslot is not preferred for that particular course or by the course’s instructor. Our system also considers and performs the assignment of rooms to courses. Each course has a list of classrooms that are suitable for it, and may only be taught in these rooms (for courses with multiple sections, including separate lab and lecture sections, each section is treated as a separate object for scheduling purposes). Additionally, rooms may be completely unavailable during specific timeslots, due to usage by parties external to the scheduling process. If a course is assigned to an unsuitable room, or to a room during a time when it is unavailable, or to two a room where another course is already scheduled, a prohibitive penalty of 10,000 is added to the total penalty. This penalty is called the unavailable-room penalty.
Formal Definition of The Problem

We formally define a course-timetabling problem as addressed in our work as a 12-tuple $(V, R, T, C, P, O, G, S_{VT}, S_{VR}, S_{RT}, d, r)$.

$V$  A finite set of events to be scheduled. In our work, the events are sections of courses.

$R$  A finite set of rooms.

$T$  A finite set of timeslots.

$C$  The conflict penalties. A function from $V \times V$ to the nonnegative real numbers that defines the conflict penalty incurred if a given pair of events are assigned to overlapping timeslots. In our system, $C$ outputs nonnegative integers, but using real numbers makes no difference in the effectiveness or behavior of our methods. $C$ must be symmetric; $\forall v_1, v_2 \in V, C(v_1, v_2) = C(v_2, v_1)$.

$P$  The proximity penalties. A symmetric function from $V \times V$ to the nonnegative real numbers that defines the proximity severity of a pair of courses. Again, our system uses integers.

$O$  Overlapping timeslots. A conflict is incurred whenever there exists a period of time during which both courses involved are taking place simultaneously. Thus, it is not enough to avoid assigning such courses to the same timeslot; they should not be scheduled to timeslots that share any period of time, which we call overlapping timeslots. $O$ is a subset of $T \times T$ containing all pairs of overlapping timeslots. It must also be symmetric relation; $(t_1, t_2) \in O$ if and only if $(t_2, t_1) \in O$. Moreover, all timeslots overlap themselves, so $\forall t \in T, (t, t) \in O$.

$G$  Gap factors. A symmetric function from $T \times T$ to the nonnegative real numbers that defines the gap factor of a pair of timeslots.

$S_{VT}$  Suitability of events for timeslots. A function from $V \times T$ to the nonnegative real numbers that defines the penalty incurred if a given event is assigned to a given timeslot. If a timeslot $t$ is perfectly suitable for an event $v$, then $S_{VT}(v, t) = 0$.

$S_{VR}$  Suitability of events for rooms. A subset of $V \times R$ containing all pairs $(v, r)$ such that event $v$ could be held in room $r$.

$S_{RT}$  Suitability of rooms for timeslots. A subset of $R \times T$ containing all pairs $(r, t)$ such that room $r$ is available for scheduling during timeslot $t$.

$d$  The unavailable-room penalty. A nonnegative real number that is incurred whenever events are assigned to unavailable rooms, including already-reserved rooms, or to rooms that are not suitable for them.

$r$  The conflict-to-proximity ratio. A number greater than 1 that captures the relative importance of the avoidance of scheduling conflicts over the creation of compact schedules. It is used while calculating the total penalty. We currently use a conflict-to-proximity ratio of 25.

A solution to a course-timetabling problem is an ordered pair $(A_{VT}, A_{VR})$

$A_{VT}$  A function from $V$ to $T$, assigning a timeslot to each event.

$A_{VR}$  A function from $V$ to $R$, assigning a room to each event.
For a solution \((A_{VT}, A_{VR})\), we define \(V_0 = \{(v_1, v_2) \in V \times V \mid (A_{VT}(v_1), A_{VT}(v_2)) \in O\}\), the set of all pairs of events scheduled to overlapping timeslots.

The objective is to minimize the following penalty function:

\[
\begin{align*}
&d[|A_{VR} - S_{VR}|] + d[|((v_1, r), (v_2, t)) \in A_{VR} \times A_{VT} \mid v_1 = v_2 \land (r, t) \notin S_{RT}] + \\
&\frac{1}{2} d[|((v_1, t_1), (v_2, t_2)) \in A_{VT} \times A_{VT} \mid (t_1, t_2) \in O \land A_{VR}(v_1) = A_{VR}(v_2)] + \\
&\sum_{(v, t) \in A_{VT}} S_{VT}(v, t) + \frac{1}{2} r \sum_{(v_1, v_2) \in V_0} C(v_1, v_2) + \frac{1}{2} \sum_{((v_1, t_1), (v_2, t_2)) \in A_{VT} \times A_{VT}} P(v_1, v_2) G(t_1, t_2)
\end{align*}
\]

The first three terms capture room unavailability: unsuitable rooms, followed by rooms unavailable at the assigned time, followed by rooms assigned to courses with overlapping timeslots. The fourth term captures the penalties for unsuitable or non-preferred timeslots. The fifth term captures scheduling conflicts, and the last term captures proximity penalties. In the third, fifth, and sixth terms, multiplication by \(\frac{1}{2}\) accounts for the fact that relevant subsets of two items will be included twice, one for each ordering. The conflict-to-proximity ratio \(r\) is used to increase the weight of the fourth and fifth terms relative to the weight of the last term. It is not applied to the first three terms, as \(d\) is already set to be prohibitively high.

### II. The Weighted-Graph Model

Many authors have modeled timetabling problems as graph coloring, e.g., de Werra (1985); Burke, Kingston, et al. (2004); and Qu et al. (2009). We use a weighted-graph model of course timetabling first developed in Kiaer and Yellen (1992) and expanded in Carrington, Pham, et al. (2007) and Wehrer and Yellen (2013). Courses in \(V\) are represented as vertices, and conflict and proximity severities as edge-weights. More specifically, each pair of vertices \((v_1, v_2)\) may have a conflict edge between them, whose weight is the conflict penalty \(C(v_1, v_2)\), and may also have a proximity edge between them, whose weight is the proximity severity \(P(v_1, v_2)\). Edges of zero weight are not included, so vertices will be non-adjacent if and only if \(C(v_1, v_2) = P(v_1, v_2) = 0\).

Furthermore, since proximity penalty reflects students or professors taking both courses, and this condition also creates scheduling conflicts, in practice a nonzero proximity severity implies at least a light conflict severity, so two vertices connected by a proximity edge will always be connected by a conflict edge. The converse is not necessarily true. This relationship between proximity and conflict edges is convenient for purposes of discussion, since the set of neighbors of a given vertex is therefore equal to the set of all vertices connected by a conflict edge.
In the graph-coloring representation, the timeslots are represented as colors, and a solution $A_{VT}$ is represented by a coloring of the graph. The weights on conflict and proximity edges are incurred when their endpoints are assigned overlapping timeslots or timeslots with nonzero gap factors, respectively. Thus, we refer to the penalty function associated with a solution as the total penalty of the coloring.

Figure 1. A three-course-timetabling problem modeled as a weighted graph. The conflict edges are marked with their conflict categories M and H, and the proximity edge is marked with its proximity severity, 4.

Figure 2. A coloring of the weighted graph from Figure 1. Conflict and proximity edges whose weights have been incurred are colored in red.

<table>
<thead>
<tr>
<th>Timeslots</th>
<th>Overlapping Timeslots</th>
<th>Distant Timeslots</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTh 8:00-9:15 AM</td>
<td>TTh 2:00-3:15 PM</td>
<td>TTh 8:00-9:15 AM</td>
</tr>
<tr>
<td>MWF 9:00-9:50 AM</td>
<td>TTh 3:00-4:15 PM</td>
<td>TTh 3:00-4:15 PM</td>
</tr>
<tr>
<td>TTh 2:00-3:15 PM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTh 3:00-4:15 PM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total Penalty:** $rH + 4 = 25(400) + 4$

**Conflict Penalty:** $H = 400$

**Proximity Penalty:** 4
The weighted-graph model as defined does not incorporate rooms, which must be assigned separately, or the associated unavailable room penalties, which must be calculated separately. However, in practice, availability of rooms is usually an easier constraint to obey than avoidance of scheduling conflicts and proximity violations. In Wehrer and Yellen (2013), each vertex is assigned a room immediately after being assigned a color. The room is chosen using heuristics based on the suitability of a room for a vertex’s neighbors. Our system uses the same approach. Since we assign rooms during the coloring process, our usage of the terms “coloring” or “partial coloring” includes the assignments of rooms, and unavailable room penalties are included in the total penalty of a coloring, unless otherwise specified. Similarly, penalties based on timeslot suitability $S_{VT}$ are not represented in the graph itself but are included in our notion of a coloring and its total penalty.

III. Graph-Coloring as a Search Problem

Our weighted-graph model allows us to view the timetabling process as a graph-coloring process. We can view graph-coloring as a state-space search. Each node in the state space is a partial coloring of the graph, including room assignments. Equivalently, each node is an ordered-pair $(A'_{VT}, A'_{VR})$, where $A'_{VT}$ and $A'_{VR}$ are functions on a subset of $V$. Since we assign rooms when assigning timeslots, the functions must have identical domains. It is important to distinguish between nodes in the search tree, which are partial colorings of the graph, and vertices in the graph itself.

We further structure the search space as a tree. Given a partial coloring, its children are all partial colorings that differ only in that one additional vertex is colored. Formally, a valid partial coloring $(B'_{VT}, B'_{VR})$ is a child node of a partial coloring $(A'_{VT}, A'_{VR})$ if $A'_{VT} \subseteq B'_{VT}$, $A'_{VR} \subseteq B'_{VR}$, and $|B'_{VT} - A'_{VT}| = |B'_{VR} - A'_{VR}| = 1$. 
Our root node, or initial state, is the uncolored graph. Since we color one vertex at each level of the tree, the height of the tree is equal to the number of vertices in the graph, or $|V|$. The leaf nodes of the tree are all at the same level, are complete colorings of the graph, and each has an associated total penalty. Since every possible coloring of the graph will be represented as a leaf node, we may frame the course-timetabling problem as a search problem: Given the uncolored graph representing the problem as the root node, find a leaf node of minimum total penalty.
Figure 4. An overview of the full search tree for a timetabling problem. At the root is the uncolored graph, and at the bottom several of the possible leaf nodes, along with their total penalties.

**IV. Priority Queue Algorithm**

Our approach to the search problem uses an algorithm based on a priority queue. It is similar to the A* algorithm used in shortest-path problems. The algorithm maintains a queue of nodes (partial colorings) kept in non-decreasing order by a priority function, which assigns a real number to every partial coloring in the search space. This priority function will determine which branch of the tree the algorithm explores next. The algorithm iterates between two distinct phases.

During the *branching phase*, the node at the front of the queue (having minimum priority function) is popped off the queue. Then, one or more children of that node are generated and added to the queue. The parent node, previously removed from the queue, is then discarded. This differs from standard A* implementations, where the parent node is retained because the path by which the solution is found is important. In course-timetabling problems, we need only find the best leaf node; the path by which we find this node does not matter, and in fact there are multiple paths that lead to identical leaf nodes.

Adding the children to the queue requires sorting them into the queue, which requires evaluating the priority function for each of them. This process is the *sorting phase*. Once all children are sorted into the queue, the branching phase may begin again, and the algorithm repeats. Viewing the problem in terms of the search tree, the sorting phase chooses a branch of the tree to explore next, and the branching phase begins the exploration of that branch by generating the children of its root node. The algorithm
begins with the uncolored graph on the queue, and iterates through the two phases until the node at the front of the queue is a complete coloring. When a complete coloring is popped off the queue, it is chosen as our solution, and the algorithm terminates.

![Figure 5. Illustration of the priority queue algorithm exploring the search tree.](image)

Since the priority function decides which branch of the tree to explore next, it should choose branches that are likely to lead to good solutions, i.e. the most promising branches. That is, promising nodes are those whose complete coloring descendents are of low total penalty. Intuitively, the priority function is a heuristic measure of how promising a node is. Thus, the algorithm constitutes a best-first search.

The algorithm also has the advantage of providing built-in backtracking. A well-designed priority function should increase as the penalties incurred by a partial coloring increase. Thus, if the node at the front of the queue has high penalties for all its descendants, these descendents will move towards the back of the queue. This allows the algorithm to “recover” from that branch of the tree and return to more promising branches.

**Performance and Computational Complexity**

Let $g(n)$ be the total penalty of a (partial or complete) coloring $n$ in the search space, and $D(n)$ denote the set of all complete colorings descended from a node $n$. If all possible children of a node are generated during the branching phase, our algorithm finds
an optimal solution provided the following condition holds: For any two nodes $m, n$ in the search space, if $g(m) > \min\{g(x) | x \in D(n)\}$, then $f(m) > f(n)$. Thus, if a suboptimal solution (leaf node) $m$ is put on the queue, it will not be selected as the solution so long as there is another node on the queue that is the ancestor of a lower-total-penalty solution. In the case where all possible children of each node are generated and added to the queue, all possible leaf nodes will have an ancestor on the queue, so this condition guarantees optimality. This condition is analogous to the condition of an admissible heuristic that guarantees the optimality of A* (see Hart, Nilsson, Raphael 1968), so we denote it as an admissible ordering.

In the best case, the algorithm completes in linear time. This occurs whenever the priority function provides a perfect ordering of the search space. Formally, we may define such an ordering as follows: A priority function $f$ provides a perfect ordering if for any two nodes $m, n$ in the search space, (1) $\min\{g(x) | x \in D(m)\} \leq \min\{g(x) | x \in D(n)\}$ implies that $f(m) \leq f(n)$, and (2) if $\min\{g(x) | x \in D(m)\} = \min\{g(x) | x \in D(n)\}$ and $m$ is at a lower level in the tree than $n$, $f(m) \leq f(n)$. If $f$ provides a perfect ordering, then the algorithm makes the optimal choice during any sorting phase. If the algorithm generates all possible children during the branching phase, then a perfectly ordered priority function will ensure that at each iteration, the algorithm moves down one level of the tree in the direction of an optimal solution. Thus, the algorithm terminates in $|V|$ iterations, and finds an optimal solution. A perfect ordering implies an admissible ordering.

Figure 6. An admissible ordering of the queue. Since the minimum total penalty among the descendents of $n$ is greater than the total penalty of the leaf nodes on the queue, it is placed before them in the queue.
Figure 7. A perfect ordering of the queue finds an optimal solution in linear time. Each discarded node in the tree is the result of a single branching phase; the algorithm has found an optimal solution in 3 iterations.

However, calculating \( \min\{g(x) | x \in D(n)\} \) requires generating all possible descendants of a node, and thus reduces to a brute-force search. Since the size of the search tree is exponential on the number of vertices, a brute-force search is not feasible for any problem of significant size. More generally, since timetabling is an NP-hard problem (Cooper and Kingston 1996), it is highly unlikely that a priority function of perfect ordering with low-order time-complexity can be found, as this would imply \( P = NP \).

Without a perfect ordering of the queue, the algorithm may take significantly longer to complete. In the worst case, the algorithm must explore the entire search tree, and therefore completes in exponential time. More specifically, since the number of children of each node is at most the product of the number of vertices and the number of colors, and the height of the tree is equal to the number of vertices, the worst-case time complexity is \( O(V^T V) \).

The space complexity of our algorithm is similar to the time complexity. Each iteration of the algorithm removes one node from the queue and adds a number of nodes to the queue. The root node has \( |V| T \) children, each of which has \( |V - 1| T \) children, and so on, such that the number of children added at each iteration is bounded above by \( |V| T \) and is on average \( \frac{|V| T}{2} \). Hence, the space complexity is \( O(V^T |k|) \), where \( k \) is the number of
iterations of the algorithm. In the worst case, the space complexity is simply the size of the full search tree $O(|V||F|^{V_{val}})$. In the best case, when the time complexity is linear as discussed above, space complexity is $O(|V|^2|F|^2)$.

Having discussed the algorithm as a whole, we will now examine in detail the two phases of the algorithm, sorting and branching. Each presents its own challenges, and an effective approach to both phases is required for the algorithm as a whole to perform well.

V. Sorting Phase

The closer a priority function is to a perfect ordering, the fewer iterations of the algorithm are required. However, even an admissible ordering is difficult to achieve in a reasonable amount of time. For instance, if our priority function is $f(n) = g(n) -$ simply the total penalty so far given our partial coloring – then the ordering is admissible, but there is no guarantee the time complexity will be significantly better than the worst case. Due to the NP-hard nature of the problem, admissible functions are either slow to compute or do not approach a perfect ordering. A priority function that illustrates the latter case is $f(n) = g(n)$. This leads the algorithm to explore large parts of the tree, forcing many iterations. In other words, there is a trade-off between computation-intensive priority functions and computation-intensive backtracking of the algorithm. In order to achieve reasonable computation time, we must forfeit the guarantee of an optimal solution.

We can demonstrate why such tradeoffs are necessary by considering the effects of a partial coloring on the penalties of its descendants. For instance, in Figure 8 below, the decision to color vertex $a$ red forces all possible leaf node descendants to incur penalties, yet the total penalty of the partial coloring at the top of the figure is zero. This illustrates the challenges of developing an effective priority function: it must predict the future effects of coloring choices on penalties, but cannot afford to explore all descendants due to the exponential number of descendants possible.
Intermediate Solution: One-Pass Coloring Algorithm

Wehrer and Yellen (2013) develop an algorithm that uses heuristics to solve course-timetabling problems in linear time. At each iteration, it assigns color and room to a single vertex. Since there is no branching, the graph is colored in a single pass. We use this algorithm to provide a quick but reasonably effective priority function. Given a partial coloring $n$, we color the remaining vertices with the one-pass algorithm to obtain a complete coloring $\text{one\_pass}(n)$ descended from $n$. We use the resulting total penalty to sort the original partial coloring in the queue; that is, $f(n) = g(\text{one\_pass}(n))$. When $n$ is itself a complete coloring, the one-pass makes no changes, so $f(n) = g(n)$. The effectiveness of this priority function is thereby tied to the effectiveness of the one-pass algorithm. In particular, as $\text{one\_pass}(n)$ is not necessarily the best descendant of $n$, an admissible ordering is not guaranteed.

A notable drawback of the one-pass-coloring-based priority function is that it biases the search process. Flaws in the one-pass algorithm will result in inaccurate ranking of nodes in such a way that our priority queue algorithm will tend to find solutions in the same region of the solution space as the one-pass coloring algorithm does. However, by incorporating branching into the coloring process, our priority queue algorithm may yet find better solutions within this neighborhood.

The heuristics Wehrer and Yellen used in their one-pass algorithm are incorporated into other parts of our work as well. The next section describes how they are used in the
branching phase of our algorithm. First, I discuss possible improvements that could be made to the sorting phase.

**Future Work: Potential Priority Functions**

It would be valuable to implement and test other priority functions than the one-pass coloring. For example, in our later discussion of the branching phase, we introduce the concept of “troublesome” vertices, vertices that are difficult to color without incurring penalties. We have various vertex-selection heuristics that rate how troublesome a particular vertex is. Summing these heuristics over all uncolored vertices would yield a measure of “troublesomeness” for the graph, and since we want to make coloring decisions that create easier coloring decisions later, the overall troublesomeness of the graph would make a reasonable priority function. However, this priority function would give all complete colorings a value of zero, clearly violating the property of admissible ordering. More generally, we would need to balance the reduction in troublesomeness produced by a particular color assignment with the penalty incurred. This could be accomplished by a weighted sum of the troublesomeness and the total penalty so far.

As an alternative priority function, the total penalty incurred so far could be summed with any approximation of the *cost-to-completion*, an expected value for the remaining total penalty from this node to its best complete-coloring descendent. Estimating the cost-to-completion is difficult, but a number of approaches could be experimented with. For instance, if one could estimate the probability of assigning a color $t$ to a vertex $v$, perhaps by comparing the penalty it would incur to the penalties of other colors, then it would be straightforward to estimate the cost-to-completion by finding the product of the probability of an assignment and the penalty an assignment would incur, summed over all uncolored vertices and all colors.

**VI. Branching Phase**

Even with a pretty good priority function, our priority queue algorithm can still do a great deal of backtracking, exploring large portions of the search tree. This makes the algorithm intractably slow due to the large size of the search tree. We can mitigate this problem if we can manage the size of the search tree. We achieve this by limiting the branching factor during the branching phase of the algorithm. If the branching factor $k$ is the number of children generated during each branching phase, the worst-case time complexity is $O(k^{v_1})$. Given the best-first nature of our algorithm, a reasonably effective priority function in combination with a small enough branching factor will allow the algorithm to complete in a reasonable number of iterations.

During the branching phase, after popping the most promising node off the front of the queue, we do not generate all of its possible children. Instead, we generate a subset of its children. We would like this subset to include the most promising of the many possible children. Specifically, given a node $n$, we would like to generate a set $S$ of its children such that $\exists m \in S$ such that $\min\{g(x) | x \in D(m)\} = \min\{g(x) | x \in D(n)\}$. That is, the
children we generate should include an ancestor $m$ of an optimal leaf node among the descendents of $n$. We designate such a subset as an optimal subset of children. If the branching phase generates an optimal subset of children, then an admissible ordering of the queue will guarantee that the algorithm reaches an optimal solution.

However, generating such a subset of children involves challenges similar to those involved in developing an effective priority function. Namely, it involves predicting which children will lead to low-total-penalty solutions. If our branching phase could generate an optimal subset of children for any arbitrary branching factor $k$, then by setting $k = 1$, we would generate a single child for each node. That child would be guaranteed to be the ancestor of an optimal solution, and hence, the algorithm would find an optimal solution in $|V|$ iterations. Thus, a polynomial-time algorithm for generating an optimal subset of children of any size would imply $P = NP$. In order to keep the branching phase reasonably fast and the branching factor reasonably small, we must accept the possibility of generating a non-optimal subset of children.

**Limited Branching via Vertex and Color Selection**

Our approach to these challenges is to generate a few children using heuristics for vertex selection and color selection. When coloring a graph, we characterize some vertices as more troublesome than others. A troublesome vertex is one that will likely create problems if it is not colored early on. For instance, if a vertex has high degree, and most of its neighbors are colored before it is, it will be difficult to select a timeslot for that vertex that does not overlap with the timeslot of one of its colored neighbors.

Information about the troublesomeness of a vertex may be captured in vertex-selection heuristics, which are often used for vertex-coloring (see Kiaer and Yellen 1992; Carrington, Pham, et al. 2007). Carrington, Pham, et al. also develop color-selection heuristics that rank possible colors in an attempt to assign the best color to a vertex given a particular partial coloring of the graph. The various heuristics depend only on properties of a vertex and its neighbors, making them fast to compute even for large graphs. Burke, Pham, et al. (2012) and Wehrer and Yellen (2013) use linear combinations to combine multiple heuristics into a single ranking of vertices or of colors. We use the same heuristics and same linear combination coefficients as Wehrer and Yellen (2013), described below. For vertices $v \in V$:

- $x_1(v)$ **Bad value of colors** for $v$. A summation of the “badness value” for all possible colors, a value which depends on suitability, room availability, and conflict and proximity severities to neighbors with an overlapping color.
- $x_2(v)$ **Conflict severity degree** of $v$ to uncolored vertices. The summation of the conflict edge weights to all uncolored neighbors of $v$.
- $x_3(v)$ **Room scarcity** of $v$. This is inversely related to the number of rooms suitable for $v$ that are unoccupied at each color suitable for $v$. It thereby captures the need to find timeslots with suitable rooms available.

The vertex-selection function $vs(v) = a_1x_1(v) + a_2x_2(v) + a_3x_3(v)$ combines these three primitive heuristics using three vertex-selector weights $(a_1,a_2,a_3)$. The most troublesome vertex is the one that maximizes $vs(v)$. Currently, we use vertex-selector weights of
to these heuristics, effective generated by this branching phase are the result one select the best colors for each vertex, and they have performed well in the context of a selection heuristics are designed to select vertices that will be optimal, but it is likely to contain promising children. The vertex- and color-selection heuristics are designed to select vertices that should be colored next, and to select the best colors for each vertex, and they have performed well in the context of a one-pass algorithm such as that in Wehrer and Yellen (2013). Since the children generated by this branching phase are the result of coloring decisions that are, according to these heuristics, effective, they will lead to good solutions in situations where the

$$(a_1,a_2,a_3) = (1,0,0),$$ so our vertex selection function reduces to the bad-value-of-colors heuristic. Bad value of colors is a sophisticated and complex heuristic, and it incorporates all constraints (timeslot suitability, room availability, conflict and proximity severities). Moreover, it was found by Wehrer and Yellen to be superior to common vertex-selection heuristics such as weighted degree. However, future research could explore the use of machine learning to find better vertex-selector weights that make use of all three primitive vertex-selection heuristics, a possibility discussed later.

Our approach for color selection is similar. Given a vertex $v$, we rank colors $t$ using a linear combination $cs(t,v) = b_1y_1(t,v) + b_2y_2(t,v) + b_3y_3(t,v) + b_4y_4(t,v)$. The “best” color for a vertex $v$ is that which minimizes $cs(t,v)$. The coefficients of the linear combination are color-selector weights $(b_1,b_2,b_3,b_4)$. The primitive heuristics used in color selection are:

- $y_1(t,v)$ The conflict penalty that would be incurred if $t$ were assigned to $v$. This includes timeslot-suitability-related penalties, $S_{vt}$.
- $y_2(t,v)$ The proximity penalty that would be incurred if $t$ were assigned to $v$.
- $y_3(t,v)$ Room scarcity relative to $v$ at $t$.
- $y_4(t,v)$ Good-to-bad switch value. Given a pre-defined threshold at which the penalty that would be incurred by assigning a color is considered “bad”, the good-to-bad switch value is directly related to the number of times that, if $t$ were assigned to $v$, a “good” color would become a “bad” color for the uncolored neighbors of $v$.

Good-to-bad switch value incorporates conflict penalties, proximity penalties, and room availability, and captures the effect that a color assignment would have on the neighborhood of a vertex. Currently, we are using color-selector weights of $(b_1,b_2,b_3,b_4) = (25,1,10,1)$.

During the branching phase of our algorithm, we use $vs(v)$ and $cs(t,v)$ to generate a limited number of children. Suppose we want to generate $k$ children of a node. We do so by selecting $i$ vertices and $j$ colors for each of those vertices, such that $k = ij$. First, we evaluate $vs(v)$ on all uncolored vertices in the node, producing a sorted list $v_1,v_2,...,v_n$ in non-increasing order. Second, we select the $i$ most troublesome vertices from the list. Third, for each $v \in \{v_1,v_2,...,v_i\}$ we evaluate $cs(t,v)$ for all colors $t$. Here we are trying to minimize the heuristics, so we produce in non-decreasing order the list $t_1,t_2,t_3,...,t_m$. For each $t \in \{t_1,t_2,...,t_j\}$, the best $j$ colors for that vertex, we generate a single child by assigning color $t$ to vertex $v$. We repeat the color selection procedure for each selected vertex $v$, re-ordering the list $t_1,t_2,t_3,...,t_m$ each time.

Choosing small positive integers $i$ and $j$ and using this procedure, we generate a small subset of the possible children of a node. There is no guarantee that the generated subset will be optimal, but it is likely to contain promising children. The vertex- and color-selection heuristics are designed to select vertices that should be colored next, and to select the best colors for each vertex, and they have performed well in the context of a one-pass algorithm such as that in Wehrer and Yellen (2013). Since the children generated by this branching phase are the result of coloring decisions that are, according to these heuristics, effective, they will lead to good solutions in situations where the
selection heuristics lead to good solutions. If we let $k = 1$, then each branching phase generates a single child by assigning the best color to the most troublesome vertex, and our priority queue algorithm reduces to the one-pass algorithm developed in Wehrer and Yellen (2013) and used as our priority function. However, when multiple children are generated, our algorithm may find a superior solution to the one found by a one-pass algorithm using the same heuristics.

Future work could produce refinements of this branching process. For example, it might be worthwhile to define a threshold $T_{cs}$ for $cs(t,v)$ at which point a child would not be generated. This would be useful because a troublesome vertex $v$ might only have one or two colors that would be a reasonable choice of assignment, and all other colors would incur high penalties. While generating children for the best $j$ colors, when the next best color has $cs(t,v)$ that exceeds the threshold, the system could generate additional children by selecting additional vertices. One might incorporate a threshold $T_{vs}$ for the vertex selection as well, and branch according the following pseudocode:

```plaintext
int i, j //given parameters
list sortedVertices = []
for vertex v in uncolored vertices:
    sort v into sortedVertices using vs(v)
int x = 0 //index of next selected vertex
int k = 0 //number of children generated so far
while k < i*j and vs(v) < T_{vs}:
    vertex v = sortedVertices[x]
    list sortedColors = []
    for color t in all colors:
        sort t into sortedColors using cs(t,v)
    int y = 0 //index of next selected color
    while y < j and k < i*j and cs(t,v) < T_{cs}:
        generate new child by assigning t to v
        y++
        k++
x++
```

Other refinements might involve varying $k$, $i$, and $j$ throughout the coloring process. For instance, it is more difficult to predict the effect of coloring decisions made early on. This suggests having a large branching factor early on to explore significantly different branches of the tree, and a small branching factor later on to save computation time. Alternatively, it may be the case that in some stages of the coloring process, vertex selection has a greater impact than color selection, but in other stages, the reverse is true. This could motivate setting $i > j$ in high levels of the tree and $i < j$ in lower levels, or visa-versa. Of course, one may want to divide the coloring process (levels of the tree) into more than two “stages.” Much experimentation would be needed in order to determine whether and how to apply such refinements.

VII. Random Problem Generation

In order to determine the effectiveness of our algorithm, especially for comparing various approaches to the challenges of the branching and sorting phases, it is useful to have a
large set of course-timetabling problems on which to test our system. Wehrer and Yellen collected data from the Rollins College Sciences Division in Fall 2009, Fall 2011, and Spring 2012, so we have three real-world problem instances for testing purposes. However, our tests are likely to be more robust given a larger set of problems, so we have written code to randomly generate problems for testing purposes.

It is not sufficient to simply generate random graphs. For testing purposes, we would like a set of graphs that are characteristic of course-timetabling problems, but it is unclear what properties define a typical course-timetabling problem. Though one could describe graph-theoretic properties of the real-world problems that we have, such as edge density, average degree, maximum degree, diameter, radius, et cetera, it is unclear whether these properties suffice to capture the characteristics of a course-timetabling problem. As such, we do not take the approach of randomly generating a graph to fit such parameters.

Instead, we randomly generate problems using techniques adapted from the field of genetic algorithms. In particular, we view problems as a set of vectors and apply random mutation and crossover to generate new problems from a seed problem. When the number of mutations is relatively small, the generated problems should remain in the same area of the space of possible graphs as the seed problem. Using a real-world problem, which by definition has the characteristics of a course-timetabling problem, we are able to produce an arbitrary number of problems with similar properties.

A Procedure for Random Generation using Genetic Techniques

We have implemented our random generation procedure in the Java programming language. It encloses course-timetabling-problem objects in wrapper objects of type WeightedGraphGeneticString, the API of which presents a vector. Each entry in the vector represents a pair of courses, and has two properties: conflict severity and proximity severity. Each of these two properties may take on one of four values: NONE, LIGHT, MEDIUM, and HEAVY. The conflict penalties for these four categories were defined in Section I. For the proximity severities, we examined the data collected from Rollins and found that the number of students expected to take two courses had been estimated in one of three discrete bins. In particular, course-pairs in light conflict were estimated to have 2 students in common, medium-conflict course-pairs 6, and heavy-conflict pairs 12. Thus, our categories of NONE, LIGHT, MEDIUM, and HEAVY were associated with proximity severities of 0, 2, 6 and 12 respectively.

Positions in the vector of course-pairs are indexed by nonnegative integers. The vector supports a point-mutation operation that selects an entry and changes either the conflict or proximity severity of that entry. To keep the changes from being drastic, mutations only change the severity to neighboring categories. If a NONE severity is mutated, it becomes LIGHT. If a LIGHT severity is mutated, then with 50% probability it becomes NONE, and with 50% probability it becomes MEDIUM, the choice being determined by a call to Java’s built-in Math.random(). Similarly, a MEDIUM severity becomes either LIGHT or HEAVY, and a HEAVY severity becomes MEDIUM.
However, because proximity severity includes only the sharing of students and professors between courses, and conflict severity includes this and other factors, the proximity severity should never exceed the conflict severity. This constraint is enforced in two ways. Firstly, a mutation that would decrease the conflict severity (e.g. from MEDIUM to LIGHT) below the proximity severity (e.g. if the proximity severity is also MEDIUM) will decrease the proximity severity as well. Similarly, a mutation that would increase the proximity severity above the conflict severity will increase the conflict severity as well.

The crossover operation is considerably more straightforward: two course-timetabling-problem vectors are given as operands, an index of crossover \( n \) is specified, and a new problem vector is returned whose first \( n \) entries have identical conflict and proximity severities to the first \( n \) entries of the first operand, and whose remaining entries have severities identical to those of the second operand.

Our random generation procedure as a whole requires three basic parameters: a seed problem, the number \( n \) of problems to generate, and the number \( k \) of mutations that should occur. Firstly, the seed problem is cloned \( n \) times, each of which is wrapped in an object representing it as a vector, as discussed earlier. Secondly, \( k \) unique positions in each vector are selected at random (no position is selected twice), and their conflict severities are mutated, after which \( \frac{k}{2} \) proximity severities are mutated in the same way. Lastly, pairs of two of the now-mutated problem vectors are selected at random. For each pair of problem vectors \((A,B)\), a crossover position is chosen at random, and two final child problems are generated, one being the crossover of \( A \) with \( B \), and the other being the crossover of \( B \) with \( A \). As per the definition of the crossover operation above, the order of the operands is significant. All random choices in the algorithm are determined by Java’s built-in pseudorandom number generator, \( \text{Math.random()} \).

When the procedure terminates, it has generated \( n \) child problems of the seed problem using \( k \) mutations each, followed by crossover. Besides varying the number of mutations \( k \), we could increase the magnitude of the changes made by feeding in the child problems as seeds for successive runs of the procedure. In genetic algorithm terms, each such iteration of the procedure would produce a new generation of problems. We could allow severities to change beyond their neighboring categories by running the procedure for two or more generations.

Our random generation code is built on top of some of the code developed for the timetabling system described in Wehrer and Yellen (2013). In particular, we use the previous system’s code to read the seed problem from a file, to write the resulting child problems to files, and for the course-timetabling-problem object that the vector representation encapsulates. The random generation code only makes changes to the conflict and proximity penalties \( C \) and \( P \); all other properties of the child problems are identical to the properties of the seed problem.
Figure 9. UML class diagram illustrating dependencies among the classes used in random generation. The classes GraphInputProcessor and WeightedGraph are from the previous system written for Wehrer and Yellen (2013). The RandomProblemGenerator, WeightedGraphGeneticString, and CoursePair classes are new.

In implementing our random generation procedure, we encountered two important challenges. Firstly, we needed to map nonnegative integer indices in the vector to unique pairs of courses in the underlying problem. Secondly, we needed to concentrate mutations on course-pairs that are already in conflict. In the remainder of Section VII, I present our solutions to each of these challenges, along with proofs of their correctness.

Mapping Vector Positions to Unique Course-Pairs

Given a problem on \( n \) courses (that is, \( n = |V| \)), there are \( \binom{n}{2} \) unique course-pairs (since \( C \) and \( P \) are symmetric, order is irrelevant). If the courses are indexed by 0,1,...,\( n - 1 \), then the vector of course-pairs is \( \langle (0,1),(0,2),..., (0,n-1),(1,2),(1,3),...,(n-2,n-1) \rangle \). We index from zero because list objects in Java index from zero. Each course-pair can be thought of as an entry in an \( n \times n \) matrix whose rows are the indices of the first course and whose columns are the indices of the second course. All of the course-pairs we need are located in the upper triangle of the matrix, excluding the diagonal. Replacing the course-pairs in this region of the matrix with their indices in the vector of course-pairs, we obtain the following matrix:
Let \( f(i) \) designate the first entry in the \( i \)th row of (the upper triangle of) \( M \). For example, \( f(0) = 0 \) and \( f(1) = n - 1 \). The number of entries in row \( i \) is \( n - (i + 1) \), so we may define \( f(i) \) recursively as \( f(i + 1) = f(i) + n - (i + 1) \). Using summation notation, we obtain

\[
f(i) = \sum_{x=1}^{i} n - x \quad i > 0.
\]

The closed form is \( f(i) = in - \frac{i}{2}(i+1) \), as can be proven by induction.

An index \( k \) for the vector of course-pairs is in row \( i \) of the matrix if and only if \( f(i) \leq k < f(i + 1) \). The derivative of \( f(i) \) is \( f'(i) = n - i - \frac{1}{2} \). From this we know that \( f(i) \) is monotonically increasing when \( i < n - \frac{1}{2} \), and monotonically decreasing otherwise. It is a concave down parabola, and all row indices \( i \) of the matrix are to the left of its maximum. So to find \( i \) for a given \( k \), we solve \( i \leq f^{-1}(k) < i + 1 \), where \( f^{-1}(k) \) is the smaller of the two roots of \( k = in - \frac{i}{2}(i+1) \). We obtain this root using the quadratic formula. Since \( i \leq f^{-1}(k) < i + 1 \), we take the floor of the root; \( i = \lfloor f^{-1}(k) \rfloor \) is the row index, and therefore first course index, of the \( k \)th course-pair.

The first entry in row \( i \) of the matrix is in column \( i + 1 \). If the \( k \)th course-pair is in row \( i \), it is \( k - f(i) \) columns to the right of the first entry in its row, so it is in column \( j = \lfloor f^{-1}(k) \rfloor + 1 + (k - f(\lfloor f^{-1}(k) \rfloor)) \), which is therefore the index of the second course in the pair.

The code in our course-pair-vector object follows the method we have just proven. Given an index \( k \), it uses the quadratic formula to obtain \( i = \lfloor f^{-1}(k) \rfloor \). From that it obtains \( j \), and returns the course-pair between courses \( i \) and \( j \).

### Probabilistic Method to Accomplish Targeted Mutation

The second difficulty we encountered during random problem generation arises from the edge density of course-timetabling problems. We found that our real-world seed problems tend to have sparse graphs; the vast majority of the possible course-pairs are not in conflict. This undermines the effectiveness of our random generation procedure: when
course-pairs are selected at random for mutation, most of the course-pairs selected will have conflict and proximity severities of NONE, so our procedure as outlined earlier will add many edges of LIGHT severity to the graph, while leaving the existing edges mostly unchanged.

We developed a solution to this challenge that does not change the high-level structure of the random generation procedure. First, we specify a fraction $p$ of the total mutations that should modify existing edges (i.e. vector entries with severity LIGHT, MEDIUM, or HEAVY). The remaining mutations will add an edge. Our solution is probabilistic, forfeiting a guarantee of exactly $k$ mutations but achieving $k$ mutations on average with the desired proportion $p$.

Suppose our graph has edge density $d$ ($0 \leq d \leq 1$). That is, there are $d \binom{n}{2}$ course-pairs in conflict. Suppose also that we want a mean of $k$ mutations, and we want a mean of $kp$, where $0 < p \leq 1$ mutations to affect course-pairs that are already adjacent. Then on average, $k(1 - p)$ mutations will add light-severity edges. We accomplish this by making $m = \frac{kp}{d}$ attempted mutations of the course-pair-vector, and if an attempted mutation would add an edge (change severity from NONE to LIGHT), we ignore the attempt with probability $1 - s$, where $s = \frac{(1-p)d}{p(1-d)}$. This will produce a mean of $kp$ actual mutations to existing edges and a mean of $k(1 - p)$ actual mutations that add an edge.

Our method is ineffective only in extreme cases that we do not expect to occur. First, if $p = 0$, then $m = 0$ and no mutations will be made. Moreover, the value of $s$ is undefined. However, $p = 0$ would entail only making changes that add an edge, which runs counter to our incentive for developing this method. Second, our calculations fail when $d = 0$, but we do not expect a seed problem with zero edges. Third, $s$ is undefined when $d = 1$, but in that case $s$ will never be used, and furthermore we do not expect to encounter a complete graph as a timetabling problem. If $p = 1$, our method correctly sets $s = 0$. Our challenge is to achieve a proportion $p$ of existing-edge mutations to total mutations, and we achieve this in the range of expected cases, which is when $1 \geq p > d > 0$. The proof of correctness of our method follows.

Let $d$, $k$, $p$, $m$, and $s$ be as defined previously. All attempted mutations to existing edges result in actual mutations, and the probability that a randomly selected course-pair has a severity greater than NONE is $d$. Hence, the mean number of actual mutations to existing edges is $kp = md$. However, an attempted mutation to a course-pair with severity NONE adds an edge with probability $s$, so the mean number of edges added is $k(1 - p) = m(1 - d)s$. The edge density $d$ is a property of the seed problem, and $k$ and $p$ are parameters of the random generation procedure, so they are given. We must find $m$.
and $s$. Solving $kp = md$ for $m$ obtains $m = \frac{kp}{d}$ . We substitute into $k(1-p) = m(1-d)s$ to obtain $k(1-p) = \frac{kp}{d}(1-d)s$. Solving for $s$ yields $s = \frac{(1-p)d}{p(1-d)}$.

We modified the simple random-generation procedure described earlier to follow this method. It takes as an additional parameter the desired proportion $p$ of existing-edge mutations to edge-adding mutations, and randomly attempts $\frac{m}{2}$ mutations to conflict severity and $\frac{m}{2}$ mutations to proximity severity. Mutations to a course-pair whose relevant severity is NONE add a light-severity edge with probability $s$, so the distribution of expected changes follows the proportion $p$. We tested our random generation code with parameters $n = 1000$ problems generated, $k = 100$ mean mutations, and various values for $p$. Our results are presented in Table 1 below, and they demonstrate the accuracy of our method in producing the desired number of changes to existing edges additions of new edges.

<table>
<thead>
<tr>
<th>Given Parameter $p$</th>
<th>Average Number of Mutations to Existing Edges</th>
<th>Average Number of Mutations that Added Edges</th>
<th>Average Actual $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>10.037</td>
<td>90.083</td>
<td>0.100</td>
</tr>
<tr>
<td>0.2</td>
<td>19.878</td>
<td>79.627</td>
<td>0.200</td>
</tr>
<tr>
<td>0.3</td>
<td>30.108</td>
<td>70.563</td>
<td>0.299</td>
</tr>
<tr>
<td>0.4</td>
<td>39.754</td>
<td>60.091</td>
<td>0.398</td>
</tr>
<tr>
<td>0.5</td>
<td>50.041</td>
<td>49.929</td>
<td>0.501</td>
</tr>
<tr>
<td>0.6</td>
<td>60.114</td>
<td>40.114</td>
<td>0.600</td>
</tr>
<tr>
<td>0.7</td>
<td>70.207</td>
<td>30.313</td>
<td>0.698</td>
</tr>
<tr>
<td>0.8</td>
<td>79.739</td>
<td>19.820</td>
<td>0.801</td>
</tr>
<tr>
<td>0.9</td>
<td>90.108</td>
<td>9.944</td>
<td>0.901</td>
</tr>
<tr>
<td>1.0</td>
<td>100.074</td>
<td>0.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 1. Results of testing the random generation code for several values of $p$. The averages given are over 1000 generated problems with the given parameter $k$ (number of mutations to make) set to 100.

**VIII. Future Work: Machine Learning**

The ability to randomly generate problems is useful not only for thorough testing of our course-timetabling system, but because future work could use it for machine learning. All machine-learning techniques require a large set of training data, which the random problem generator provides. Here I discuss several ways in which machine learning could be used to make improvements on our current work.
Parameter Optimization via Genetic Learning

Firstly, machine learning could be used to optimize the parameters of our algorithm. Beyond the course-timetabling problem to solve, our algorithm requires several parameters. Both the branching phase and the one-pass priority function require three vertex-selector weights \((a_1, a_2, a_3)\) and four color-selector weights \((b_1, b_2, b_3, b_4)\). The “bad value of colors” vertex-selection heuristic and the “good-to-bad switch value” color-selection heuristic both require thresholds for “bad” conflict and proximity penalties. The “conflict severity degree” vertex-selection heuristic requires two multipliers for weighting medium and heavy severities. In the future, if the branching phase were modified to use thresholds when selecting the appropriate number of vertices and the appropriate number of colors for each vertex, it would involve two additional parameters, thresholds \(T_{vs}\) and \(T_{cs}\). Alternative priority functions might require additional parameters as well.

Our current values for our parameters are taken from Wehrer and Yellen (2013), in which they were chosen by testing a small number of possible parameter combinations on the real-world problem for the Rollins Science Division in the fall semester of 2009. Using the randomly generated problems as training data, we may be able to find more effective values for these parameters. In particular, this could be accomplished using a genetic algorithm.

Optimization of the parameters lends itself to genetic machine learning for two reasons. First, it is straightforward to represent the parameters as a gene, in particular a vector of numbers. Point mutations in the gene would correspond to varying a single parameter, increasing or decreasing the corresponding vector entry by a random amount. The range of increase or decrease could start large and shrink over successive generations, so that the algorithm would begin by exploring a large solution space and end with hill-climbing in particular areas of the solution space. The crossover operation for a vector of numbers is equally straightforward.

Second, we have a well-defined and fast fitness function. All of the parameters for our current version of the priority queue algorithm are used in similar ways in the one-pass algorithm. For problems with about 100 vertices, which is the size range of the Rollins problems and therefore of the problems generated by using them as seeds, the one-pass algorithm completes in a matter of seconds. Given a vector of parameters, we would apply the one-pass algorithm with these parameters to a set of randomly generated problems. Averaging the total penalty over these problems yields a fitness function to be minimized. In order to avoid over-fitting, a new set of random problems could be used every few generations of the genetic algorithm.

Parameter Variation throughout the Coloring

As discussed in Section VI, it may be worthwhile to explore varying the number of children generated, or emphasizing one or the other of vertex- and color-selection, at different points in the coloring process. Similarly, the optimal vertex- and color-selector
weights may vary depending on the point within the coloring process. Genetic parameter optimization could be expanded to explore these possibilities. In the simplest case, the coloring process would be divided into a number of discrete “stages.” Having a set of vertex- and color-selector weights for each stage would simply entail a longer parameter-vector to optimize. Alternatively, vertex- and color-selector weights could vary as functions of the number of vertices colored, and the characteristics of the functions (e.g. slope of a linear function) could be encoded in the parameter vector to be optimized.

Variation of the number of children generated, or the numbers of vertices and colors selected, could be optimized in a similar way. However, the fitness function would have to use the priority queue algorithm instead of the one-pass algorithm, so the computation time would be significantly longer. It may not be feasible on large sets of training data for many generations. An alternative approach would be to use neural networks. In particular, during the priority queue algorithm itself, feedback about the children generated according to a set of parameters could immediately tweak those parameters for the next iteration of the algorithm. For example, suppose that a particular primitive heuristic dominates the decision to select a particular vertex or color. The value of the priority function for the resulting children, relative to other children of the same node, might justify increasing or decreasing the selector weight on that heuristic.

IX. Implementation of a Course-Timetabling System in Java

I have implemented our priority queue algorithm within a course-timetabling system written in the Java programming language. The work for Wehrer and Yellen (2013) included the implementation of a course-timetabling program with a graphical user interface and file input/output abilities. Beginning with this codebase, I made significant high- and low-level changes to implement the current version of the priority queue algorithm, in addition to implementing random problem generation as discussed earlier.

The previous system was built around the one-pass algorithm. It represents course-timetabling problems and their associated solutions or partial solutions with instances of the WeightedGraph class. The WeightedGraph class is vastly interconnected with the supporting Vertex, Edge, and Color classes. Information about the state of a WeightedGraph is spread out between these classes and duplicated in many places. For example, adjacency information is stored in the WeightedGraph object, which has a list of all Edges; in the Vertex object, which has a list of incident Edges; and in the Edge object, which stores a reference to each of its Vertex endpoints.

The previous codebase did not include a great deal of abstraction or generalization. WeightedGraph, Vertex, and Edge objects include details – which instructors are teaching what courses, the department that each course belongs to, the times and days included in each timeslot, et cetera – that are specific to the constraints of a particular problem, rather than a generalization of the structure of a class of problems. Thus, modifying the code to represent scheduling of different types of events under different
circumstances would require multiple changes throughout these and associated classes. In short, the code of the previous system is not very reusable or extensible.

Furthermore, the one-pass algorithm itself is not implemented on top of the `WeightedGraph` model of the problem and its coloring state. Instead, though much of it is implemented in the separate `Timetabler`, `VertexSelector`, and `ColorSelector` classes, it is deeply integrated into the classes of the model. For instance, the choice of room assignment is made in the `Vertex` class when the method assigning a color to a vertex is called, and many primitive vertex- and color-selection heuristics are implemented as methods of the `Vertex` class or of the `PenaltyPairList` class. The latter class is a mechanism used by the one-pass algorithm to track the state of conflict and proximity penalties, but is used within the `WeightedGraph`, `Edge`, and `Vertex` classes as well, further duplicating the storage of state information within a `WeightedGraph` and associated objects. This tight integration between model/problem state and algorithm state weakened the extensibility of the code. Significant rewriting was required to support a different coloring method such as our priority queue algorithm.

Thus, while writing new code to implement the priority queue algorithm, and integrating it into the existing codebase, I introduced a new high-level structure with the aim of a cleaner architecture that would more easily support future modification and extension. A simplified UML diagram of the resulting program architecture is presented below. The most significant change is the introduction of the `Model` interface, which encapsulates a problem state with an API similar to our formal definitions of timetabling problem and solution presented at the end of Section I. Any class may implement this interface, and so any means of representing a course-timetabling problem and solution could be employed. The `Model` interface also uses Java’s generic typing feature to introduce further abstraction. It represents a course-timetabling problem on a set of `Events`, `Rooms`, and `Timeslots`, abstract interfaces that can be applied to any concrete Java class. Hence, the code could be easily reused to schedule different types of events. The priority queue algorithm is built on top of the `Model` interface, and does not know or care what `Model` is used. Besides `Model`, it depends only on the abstract `PriorityFunction` and `NodeExpander` interfaces, so changes to the sorting phase or to the branching phase can be made easily. One need only write a new concrete class that exposes the appropriate interface, and pass an instance of it to the `PriorityQueueAlgorithm` constructor.
Figure 10. UML class diagram illustrating the structure of the new code, and its integration with the previous code. Thin arrows represent dependencies, and thick arrows represent inheritance, in this case between abstract interfaces and the concrete classes that implement them.

Beyond these high-level changes, one of the most significant changes made to the system was the introduction of model cloning. In the one-pass algorithm, only a single instance of the problem, including the solution, is needed. This takes the form of a single WeightedGraph instance and all of the objects it encapsulates. The one-pass algorithm, makes color assignments propagate them throughout this system of objects. A single system of interconnected objects models not only the problem itself, but also the complete coloring solution and each partial coloring along the way. However, the priority queue algorithm keeps multiple partial colorings in the queue, and changes to one such node should not affect other nodes. Thus, the Model interface expects Model objects to be able to make copies of themselves, and expects timeslot or room assignments to these copies to have no effect on the original or on other copies. This mechanism is used in NodeExpander objects to generate children of a node. Cloning is also necessary for random generation, and each clone must be able to accept changes to the conflict and proximity severities without affecting the original or other clones.

In order to support these uses of cloning, I spent a great deal of time adding cloning capability to the WeightedGraph class. The duplication of data, and the tight integration
of the various supporting classes with little abstraction, made this a challenging and time-
consuming process. Since changes in state could propagate throughout the whole system
of objects, I had to write code that builds a completely new system of objects with no
references to any object in the original system. Because data is duplicated in multiple
places in multiple ways, naïve changes to data can break synchronization between these
duplicates and with other data they affect, resulting in an illegal state. Hence, the copying
of data from the original system of objects to the clone must be done in careful order.
Fortunately, the existing code for reading a WeightedGraph from a file provided me with
a template for this procedure.

**Current Challenge: Memory Space Conservation**

At this point, our testing of the priority queue algorithm has been somewhat limited. Our
algorithm is finished and working, and has been verified to produce an identical result to
the one-pass algorithm when the branching factor is set to 1. It produced comparable
results to the one-pass algorithm with branching phase parameters \( i = 1 \) and \( j = 2 \). That
is, one vertex was selected and two colors selected for that vertex. However, when setting
\( j > 2 \) or \( i > 1 \), we were unable to run the algorithm to completion without causing a heap
overflow. Though it is expected for the size of the queue to grow large, the main culprit
in our case is likely to be the significant amount of memory space taken up by each node
in the queue. This large memory size is due to the duplication of data throughout the
system of objects associated with a WeightedGraph, and the need to create new copies of
all relevant objects when cloning a node. To give you a sense of the size of each node:
with the Java VM heap size set to 128 megabytes, the system consistently crashes with
around 200 WeightedGraph instances in memory, each instance representing a problem
with 96 courses. Finishing the algorithm with \( i = 1 \) and \( j = 2 \), on a graph of 96 vertices,
required setting the maximum JVM heap size to two gigabytes.

Moreover, when heap space is running low, the branching and sorting phases of the
algorithm change from taking under 100 ms each to lasting several seconds and
eventually over a minute. This is most likely due to increased garbage collection and to
the difficult search for contiguous memory blocks in which to store the many
ArrayLists used in a WeightedGraph and its associated objects. The sorting phase is
also affected, because in order to perform a one-pass coloring without modifying the
original, the priority function must first clone the node that it is evaluating.

**Solution: Multi-Layer Graph Model**

To address the issue of memory space, I am currently implementing a new, lightweight
Model on top of the existing WeightedGraph class. This new model is the
MultiLayerGraphModel class, and represents a problem and partial or complete solution
using a number of simple graphs (simple referring to having at most one edge between
any two vertices). It is built on top of several utility classes that store simple weighted or
unweighted graphs using adjacency matrices. These classes use generic typing, allowing
the vertices of the graph to be objects of any class.
Following along with the formal specification of a problem instance given in Section I: \( C \) and \( P \) are each represented by a weighted graph on the events \( \mathcal{V} \). Timeslot overlaps \( \mathcal{O} \) are represented by an unweighted graph on \( \mathcal{T} \), with edges between pairs of timeslots that overlap. \( G \) is similarly represented, albeit with a weighted graph whose edge-weights are the gap factors. \( S_{VT} \) is represented by a weighted bipartite graph, whose vertex bipartition consists of sets \( \mathcal{V} \) and \( \mathcal{T} \). \( S_{VR} \) and \( S_{RT} \) are also represented by bipartite graphs, but they are unweighted bipartite graphs with edges between suitable pairs. In all weighted graphs, vertex-pairs that would have an edge-weight of zero are considered to be non-adjacent.

The (partial) coloring information \( A_{VT} \) and \( A_{VR} \) is represented by two assignment graphs. These are bipartite graphs in which the vertices of the first side, \( \mathcal{V} \), of the partition are adjacent to at most one vertex in the second side. Edges represent assignments of events to rooms or timeslots. Finally, to calculate the availability of rooms, the model maintains two bipartite graphs: a room reservation graph \( A_{RT} \) indicating which rooms are reserved at which timeslots, and a current suitability graph \( S'_{RT} \). In the room reservation graph, vertices \( r \) and \( t \) are adjacent if and only if \( \exists v \in \mathcal{V}, A_{VT}(v) = t \land A_{VR}(v) = r \). In the current suitability graph, vertices \( r \) and \( t \) are adjacent if and only if \((r,t) \in S_{RT} \land \neg (\exists (r',t') \in A_{RT}, (t,t') \in \mathcal{O})\). Thus, room \( r \) could be reserved during timeslot \( t \) without penalty if and only if \( r \) and \( t \) are adjacent in \( S'_{RT} \).

Figure 11. An example of a suitability graph, room reservation graph, overlap graph, and the resulting current suitability graph. Note that \( r_2 \) is no longer suitable for \( t_3 \) because it is reserved during the overlapping timeslot \( t_2 \).

The multi-layer graph model also creates a clear separation between the specification of the problem, which we call inherent data, and the specification of the solution or partial solution, which we call dynamic data. The inherent data is constant throughout the coloring process, so the priority queue algorithm need only maintain one copy in memory. The dynamic data changes with each partial coloring, and includes the assignment graphs, room reservation graph, and current suitability graph. Inherent data is initialized using an uncolored WeightedGraph object. The WeightedGraph object is wrapped in a MultiLayerGraphModel object. The MultiLayerGraphModel class
contains all dynamic data and extends the `InherentMultiLayerGraphModel` class, which contains all inherent data. When a `MultiLayerGraphModel` object clones itself, it only allocates space for new dynamic data. The clone will have new assignment, room reservation, and current suitability graphs, but will contain pointers to the inherent data of the original `MultiLayerGraphModel`. Changes to the dynamic data will have no effect on the inherent data, and the `InherentMultiLayerGraphModel` class has no dependencies on the `MultiLayerGraphModel` class, so it can be used to represent a problem without any solution information.

![Figure 12. Simplified UML class diagram of the MultiLayerGraphModel and related classes.](image)

The multi-layer graph model has several advantages. First, it avoids duplication of data. Because the graph objects use adjacency matrices instead of edge objects, each piece of information about connectivity, conflict/proximity severities, room and timeslot assignments, and so forth is stored in a single place. Hence, the state can be modified more easily, and the code restructured more easily in the future. Second, the clear separation of inherent and dynamic data is cleaner and more extensible. Of greater practical importance, the cloning of only the dynamic data will result in immense reduction of necessary memory space. Third, the model is abstract and generically typed. We currently use a `WeightedGraph` object, because the previous system provides code to create these objects with a GUI and to read them from and write them to a text file. However, a `MultiLayerGraphModel` could be constructed from any object that implements the `Model` interface and would use that object to initialize both inherent and dynamic data. Finally, since all state information is represented in the edges and edge-weights of the multi-layered graph, it is possible that in future work, connectivity and
path information about this graph might lead to more sophisticated heuristics for vertex selection, color selection, the priority function, or other purposes.

The multi-layer graph model is fully developed from a design standpoint, but its implementation in our Java timetabling system is not yet finished and tested.

**X. Conclusion and Future Research**

The priority queue algorithm that we have developed provides a flexible framework for automated timetabling. It lends itself to the application of many optimization techniques, particularly machine learning and heuristics. Our system for random problem generation provides a powerful tool for further research on the problem. However, significant further testing of the priority queue algorithm is needed before we can be confident in the quality of the solutions it produces. Moreover, defining an effective priority function remains a difficult challenge, and further research on this aspect is needed in order to establish the priority queue algorithm as a competitive approach to course-timetabling.

The most immediate direction for future work would be the implementation of the multi-layer graph model. The small memory footprint of the model is crucial if the priority queue algorithm is to be subjected to extensive testing. Second, since vertex- and color-selection procedures like those used in the branching phase are well-studied in other work, the development of a more sophisticated priority function presents itself as a novel question for future study. However, the incorporation of machine learning into various aspects of the system, as discussed in Section VIII, is a more straightforward line of work that may yield more immediate results.

**Acknowledgements**

I thank Anthony Wehrer ’11 for the research and programming that laid the foundation for this project. I thank the faculty in the Science Division at Rollins College for participating in the testing of the previous system, and am particularly grateful to the following faculty members: Dr. Christopher Fuse, director of the Student-Faculty Collaborative Scholarship Program, for generously supporting my presentation at the 2014 National Conference on Undergraduate Research in Lexington, KY; my thesis advisor and timetabling expert, Dr. Jay Yellen, for his dedication, support, and high standards of grammar and diction; Dr. Jennifer Seitzer for introducing me to the field of artificial intelligence, including the techniques used in our system; and Dr. Julie Carrington and Dr. Julie Anderson for nurturing my enthusiasm for elegant software design.
References


Carrington, J. R., Pham, N., Qu, R. & Yellen, J. (2007). An enhanced weighted graph model for examination/course timetabling. In R. Barták et al. (Eds.), *Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group* (pp. 9-16). Prague: Charles University.


