Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Administrador de Conflitos de Horário

Eduardo Brancher Urenha

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Prof. Carlos Eduardo Ferreira

São Paulo
2023

Resumo

Eduardo Brancher Urenha. **Administrador de Conflitos de Horário**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O problema de construção de horários trata de definir a distribuição de um conjunto de aulas no tempo disponível para a semana. Essa distribuição deve levar em conta a disponibilidade de recursos necessários para aulas, como professores e salas, e garantir que nenhum recurso seja utilizado em duas aulas ao mesmo tempo, pois isso seria impossível. O problema é relativamente intratável e apresenta subproblemas NP-difíceis. Desenvolveu-se uma solução capaz de medir o valor relativo de um dado horário do ponto de vista dos alunos, por meio do cadastro da intenção de matrícula dos alunos e contagem da quantidade de conflitos de horários presentes. O programa desenvolvido também permite a alteração pontual do horário para que sugestões de mudança de horário sejam comparadas com uma métrica objetiva. Espera-se minimizar a quantidade de mudanças de horário que os Representantes Discentes (RD) precisam fazer durante a elaboração final da grade horária.

Palavras-chave: Administrador. Conflitos. Horários.

Abstract

Eduardo Brancher Urenha. **Timetable Conflict Manager**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

The timetable construction problem deals with distributing a set of classes in the available time for a week. This distribution must take into account the availability of resources needed for each class, like professors and rooms, and guarantee that no resource is used by two classes at once, for this would be impossible. The problem is relatively intractable and presents NP-hard subproblems. A solution is developed which is capable of measuring the relative value of a given timetable from the students' viewpoint, through the registering of enrollment intentions for students and counting of the amount of time conflicts that are present. The developed program also allows for localized alteration of the timetable so that changes to the timetable may be compared with an objective metric. We expect to minimize the amount of timetable changes that Students' Representatives (SR) must make during the final construction of a timetable.

Keywords: Administrator. Conflicts. Timetables.

Sumário

Introdução

Definição do Problema

O problema de construção de horários é um problema que tem diversas formas de ser apresentado, cada uma com características ligeiramente diferentes, e não há uma única definição universalmente utilizada. Nesse trabalho, consideramos um subconjunto específico de problemas de construção de horário, definido da seguinte forma: Considera-se que uma instituição educacional oferecerá um conjunto de C cursos para um conjunto A de alunos. Cada um dos cursos $c_i \in C, i = 1, ..., |C|$ necessita de um conjunto de recursos. Os recursos são elementos de dois conjuntos: um conjunto de professores P e um conjunto de salas S. Cada curso precisa de 1 ou mais professores $p_j \in P, j = 1, ..., |P|$ e de 1 ou mais salas $s_k \in S, k = 1, ..., |S|$. Além disso, um curso c_i precisa ocorrer em uma certa quantidade de janelas de tempo. Uma janela de tempo é definida como um intervalo específico de tempo em um dado dia da semana. Por exemplo, segunda-feira, das 14:00 às 15:00. Uma restrição fundamental do problema é que um recurso qualquer não pode ser utilizado por 2 ou mais cursos que ocorram na mesma janela de tempo, pois isso seria impossível - um professor teria de dar duas (ou mais) aulas ao mesmo tempo, e uma sala usada para duas (ou mais) aulas ao mesmo tempo. Outras restrições podem ser definidas a depender da particularidade de cada instituição, como por exemplo classes de recursos (uma matéria c_i aceita apenas um subconjunto específico de elementos de P ou S, como professores especialistas e laboratórios ao invés de salas genéricas), cargas horárias mínimas e máximas para cada professor, ocupações máximas de salas, minimização de conflitos para os alunos (por exemplo, casos em que um aluno precisaria assistir a duas aulas ao mesmo tempo), entre outros. Note que os alunos não são recursos, portanto tais conflitos são permitidos, embora indesejáveis.

Um horário aceitável, portanto, é um horário que associa para todo curso c_i os recursos e janelas de tempo necessários, de modo que nenhum par de cursos que compartilhe algum recurso compartilhe também alguma janela de tempo, além de respeitar outras restrições rígidas que possam ser formuladas.

Motivação do Trabalho

Em primeiro lugar, descreve-se brevemente como se dá o processo de definição de horários do Bacharel em Ciência da Computação (BCC) no Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP). Em primeiro lugar, a carga de matérias que serão oferecidas em um semestre e os professores que lecionarão essas disciplinas

são definidos. Em seguida, a comissão de horários do BCC decide em quais horários na semana cada aula ocorrerá, tomando certos cuidados iniciais correspondentes a uma série de restrições e heurísticas estabelecidas no departamento.

Depois que essa grade inicial é definida, ela é publicada para apreciação dos alunos. Esses alunos tem um prazo, em geral de uma ou duas semanas, para requisitar alterações no horário aos seus representantes discentes (RD). Essas alterações são propostas pelos alunos como anotações na própria planilha de horários que é compartilhada, e são discutidas pelos alunos por meio de comentários sobre as anotações na planilha.

O formato de elaboração dos pedidos de mudança de horário descrito no parágrafo anterior tem alguns problemas sérios. Em primeiro lugar, os alunos só recebem notificações sobre atualizações em algum tópico de mudança se fizeram um comentário sobre a anotação relevante. Por exemplo, se há uma mudança sendo discutida sobre uma dada matéria A, apenas alunos que comentaram sobre essa anotação de mudança recebem e-mails informando-os de alguma atualização na discussão. Isso significa que alunos que não estão informados da mudança, mas que podem estar interessados nela, ou ser contrários a ela, continuarão sem ser informados até que proativamente chequem todos os comentários na planilha. Em segundo lugar, um aluno pode verificar a planilha e constatar que os horários e mudanças em discussão são satisfatórios, e mesmo assim estar desatualizado, pois no dia seguinte novos tópicos podem ser propostos. Em conjunto, os problemas significam que os alunos têm de verificar a planilha diariamente se querem ter voz sobre todas as mudanças em discussão, algo que simplesmente não é feito pela maioria.

O efeito final dos problemas mencionados é que invariavelmente, quase todo pedido de mudança é contestado depois de ser feito. Basicamente, alunos que estavam satisfeitos com os horários publicados não tem nenhum incentivo para checar a planilha consistentemente e, desse modo, se ocorre uma mudança nos horários que é insatisfatória para os que estavam satisfeitos, essa mudança é contestada. Não é raro que mudanças sejam revertidas em razão de um número de contestações maior que a quantidade inicial de alunos que pediram a mudança. O processo acaba por gerar uma grande quantidade de retrabalho para os RDs e os professores que porventura tenham realizado mudanças na grade para atender a requisições de alunos. Abaixo disponibilizamos um fluxograma que ilustra o processo decisório de alterações na grade.

Para auxiliar os representantes no processo de alteração dos horários, inicialmente se propôs desenvolver um programa que fosse capaz de receber um conjunto de restrições dos professores, um conjunto de intenções de matrículas dos alunos, e criasse um horário que respeitasse todas as restrições e gerasse o mínimo de conflitos de horários para as intenções dos alunos que foram recebidas. Entretanto, após uma pesquisa bibliográfica que será detalhada na próxima seção, descobriu-se que o problema é relativamente intratável, sendo NP-difícil, e que em geral não é possível modelar todas as restrições que podem surgir na prática.

Em razão da relativa complexidade das soluções existentes para a solução de um problema arbitrário de construção de horários examinadas na revisão, tomamos a decisão de limitar o escopo do projeto à produção de uma solução que fosse capaz de avaliar o efeito de realizar alterações em um horário dado, ao invés de construir um horário. Uma solução

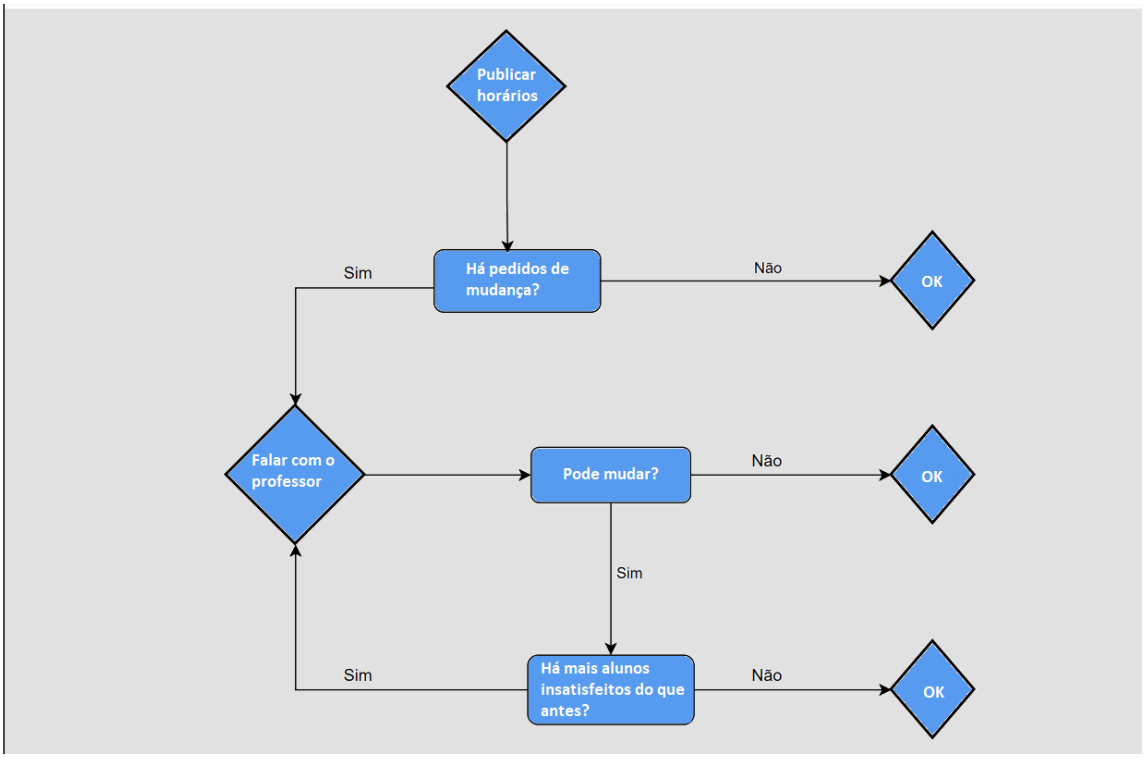


Figura 1: *Fluxograma de alterações na grade horária do BCC.*

desse tipo seria ainda muito útil para o usuário alvo do trabalho, o representante discente (RD), pois via de regra ele não tem muita agência na produção de uma determinada grade, mas a existência de uma solução capaz de avaliar o valor relativo de uma mudança proposta será útil para que o RD decida quais mudanças devem ser levadas adiante e discutidas com o corpo docente.

Adicionalmente, menciona-se ainda o fato de que nenhuma das soluções discutidas na nossa revisão é plenamente satisfatória, sendo que todas podem necessitar de ajuste manual do usuário depois que o programa devolve a melhor solução que pôde obter. Desse modo, nosso trabalho não seria supérfluo mesmo na presença de um construtor automático de horários, pois define uma métrica para quantificar o valor de alterações na grade e permite o ajuste fino de soluções que poderiam ser produzidas por outros métodos.

Estrutura da Monografia

O presente trabalho se divide em 3 capítulos, além dessa introdução: uma revisão bibliográfica em que aspectos do problema são discutidos em detalhes, como sua intratabilidade, e são relatados alguns métodos de solução publicados na literatura pra certas versões do problema; uma seção dedicada aos resultados e discussão em que explicamos a solução que foi desenvolvida para refinar um horário construído, e uma conclusão em que discutimos o impacto do trabalho e direções futuras de desenvolvimento.

Capítulo 1

Problema de Construção de Horários

A bibliografia sobre o estudo de problemas de construção de horários, em inglês *timetable construction problems*, começa a surgir nos anos 1960, em particular com a definição inicial do problema por (Gotlieb, 1962). Embora existam versões do problema que podem ser resolvidas em tempo polinomial, em geral com exclusão de restrições, como resumido por (Cooper e Kingston, 1995), a versão com restrições de horário sobre os alunos e professores é um problema NP-difícil. Usamos um argumento inicialmente proposto por (Welsh e Powell, 1967) para problemas de escalonamento de tarefas. Tais problemas são equivalentes a colorir os vértices um grafo de modo que vértices vizinhos não compartilhem uma cor, que Karp mostrou ser NP-Completo (Karp, 1972). Formalmente, o problema da coloração de um grafo é definido da seguinte forma: Considere o conjunto de vértices V de um grafo. Define-se uma função $f : V \longrightarrow \mathbb{Z}$ tal que um número inteiro representa uma cor. Por exemplo, se v_i é um vértice, e $f(v_i) = 4$, o vértice v_i está associado com a cor 4. O problema da coloração então é: Dado um número de cores k, é possível associar a cada vértice do grafo uma dessas cores, de modo que para toda aresta definida por $(u, v), u \in V, v \in V$, tenhamos $f(u) \neq f(v)$? - em outras palavras, que os vértices adjacentes tenham cores distintas. Se $k = 2$, o problema pode ser resolvido em tempo polinomial. Entretanto, para $k > 2$, é NP-difícil. Dada uma solução para o problema da coloração, é possível usá-la para obter um horário aceitável para um caso particular do problema de construção de horários. O argumento foi detalhado por (Cooper e Kingston, 1995) e será reapresentado aqui.

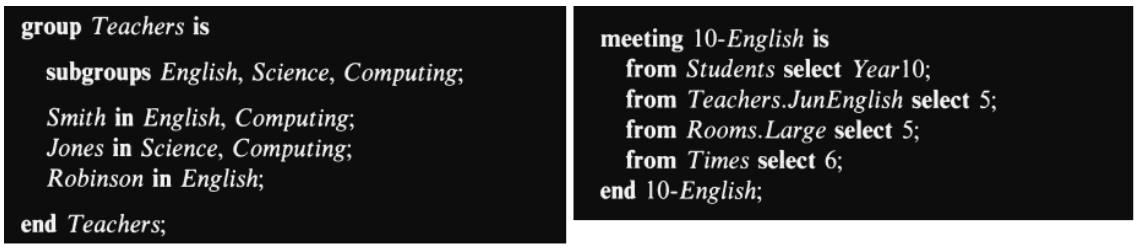
Em essência, são descartadas algumas restrições dentre as restrições originais do problema. Assume-se que o número de salas e professores disponíveis é suficiente para que todas as disciplinas sejam distribuídas em k horários, $k > 2$, e que há um certo número de aulas e de alunos. Adicionalmente, cada aluno pretende se matricular em exatamente duas disciplinas. Desse modo, as disciplinas podem ser representadas pelos vértices de V. Cada aresta $(u, v), u \in V, v \in V$corresponde a um aluno, que pretende cursar exatamente as duas disciplinas (u, v). Nesse caso particular do problema, uma coloração de um grafo em que vértices vizinhos não compartilham uma cor equivale a uma distribuição de horários em que não há conflito para nenhum aluno: cada cor corresponde a uma janela de horário.

Assim, se todo par (u, v) atender à condição $f(u) \neq f(v)$ do problema da coloração, nenhum aluno terá de assistir a duas aulas que ocorrem ao mesmo tempo. Por exemplo, seja (u, v) uma aresta qualquer. Se $f(u) = 3$ e $f(v) = 4$, a disciplina u ocorre na janela de horário 3 e a disciplina v ocorre na janela de horário 4. Portanto, o aluno (u, v) não tem conflito de horário. Desse modo, esse subproblema que considera restrições apenas sobre os alunos é redutível para o problema de colorir um grafo com k cores. Assim podemos ver que mesmo um subproblema da construção de horários em que restrições são ignoradas é NP-difícil. Note que podemos modelar as arestas como professores ou salas igualmente, para garantir que um professor ou sala não seja utilizado por duas aulas ao mesmo tempo. Nesse caso, o subproblema em que cada professor dá apenas duas aulas por semana (ou que cada sala comporta apenas duas aulas por semana) já é NP-difícil.

Um outro exemplo da intratabilidade do problema pode ser percebido quando está se decidindo a carga horária por professor ou a distribuição de turmas em salas. Em ambos os casos, (Cooper e Kingston, 1995) notam que o problema é um caso particular do problema BIN-PACKING, como definido por (Garey e Johnson, 1979), que pergunta se é possível distribuir um conjunto de $O = o_1, ..., o_n$ objetos, cada um com um determinado tamanho $S(i), i \in 1, ..., n$ em k caixas, todas com capacidade C, de modo que nenhuma receba itens cuja soma dos tamanhos exceda sua capacidade, e todos os itens sejam distribuídos, e k seja mínimo. Esse problema é NP-difícil (Garey e Johnson, 1979). Considere que queremos distribuir O disciplinas entre k professores. Então, cada professor terá um limite de horas-aula por semana (uma capacidade C) e teremos $O = o_1, ..., o_n$ disciplinas, cada uma com tamanho $S(i), i \in 1, ..., n$. Se tivermos uma solução para BIN-PACKING, teremos uma função $f : O \longrightarrow 1, ..., k$ que determina qual professor estará associado a qual disciplina (qual caixa recebe qual objeto). Para toda matéria associada ao mesmo professor, damos janelas de tempo disjuntas, de modo que as disciplinas que compartilham um professor não compartilham nenhuma janela de tempo. De modo que temos que uma construção de horários que ignora todas as restrições exceto a capacidade dos professores e a quantidade de horas-aula das disciplinas é redutível para BIN-PACKING. Naturalmente, podemos fazer a mesma redução com outros recursos como salas, e tamanho de turmas.

Existem soluções para versões do problema com conjuntos definidos de restrições, que se utilizam de diversas heurísticas para obter soluções aceitáveis. Uma solução desse tipo foi obtida por (Cooper e Kingston, 1993), na qual uma gramática é definida para expressar um certo conjunto de restrições e associações entre recursos disponíveis, que são em seguida distribuídas no tempo. Primeiro, a gramática que descreve os recursos disponíveis é formulada para aquela instituição.

Em seguida, para cada tempo t, um grafo bipartido é construído. Uma das duas partições corresponde aos recursos necessários para aquele tempo e a outra partição representa os recursos disponíveis. Uma aresta sai de cada recurso para os recursos necessários que podem ser satisfeitos por aquele recurso. Por exemplo, se Carlos é professor de Computação, Zara é professora de matemática e Yoshiharu é professor de computação e matemática, haverá uma aresta entre o recurso necessário Professor de Matemática e o recurso Zara, uma aresta entre Professor de Computação e Carlos, e uma aresta entre Yoshiharu e Professor de Computação e Professor de Matemática. As arestas representam o fato de que um recurso pode satisfazer uma determinada necessidade.



```
group Teachers is
   subgroups English, Science, Computing;
   Smith in English, Computing;
   Jones in Science, Computing;
   Robinson in English;
end Teachers;
```

```
meeting 10-English is
   from Students select Year10;
   from Teachers.JunEnglish select 5;
   from Rooms.Large select 5;
   from Times select 6;
end 10-English;
```

Figura 1.1: *Exemplo da gramática definida por (Cooper e Kingston, 1993). São definidos grupos de recursos como professores e definidas reuniões para cada disciplina que consomem uma quantidade de tempo e recursos. Fonte: (Cooper e Kingston, 1993).*

Um grafo bipartido como descrito acima é dito viável se para aquele intervalo de tempo t, pode-se encontrar um pareamento entre as duas partições do grafo. Um pareamento é um subconjunto das arestas tal que cada recurso necessário tem pelo menos uma aresta incidente, e cada recurso disponível tem exatamente uma aresta incidente. Um tal pareamento representa que há recursos suficientes para cobrir a necessidade no tempo t. Portanto, as matérias no tempo t podem ocorrer simultaneamente. Como um pareamento pode ser verificado em tempo linear e o grafo estendido em tempo linear, esse método constitui uma checagem eficiente se uma associação de um conjunto de matérias em um tempo é factível.

Finalmente, para cada grafo conforme construído acima é associado um proto-tempo, um número inteiro que representa uma janela de tempo indeterminada. Embora o proto-tempo não tenha horário definido, ele representa um tempo em que as disciplinas no grafo podem ocorrer simultaneamente: Haverá recursos suficientes para todas as disciplinas que alocamos para aquele proto-tempo, sem decidir se esse proto-tempo será segunda-feira às 8:00 ou sexta às 16:00. O procedimento do algoritmo, então, é estender o grafo bipartido de recursos com os recursos de uma dada matéria para aquele proto-tempo t. Se houver um pareamento, associar uma matéria àquele proto-tempo é possível, pois haverá recursos suficientes. Uma vez que toda matéria tem um proto-tempo definido e não há conflitos, os proto-tempos são traduzidos para tempos de verdade. Vários conjuntos de tempos podem ser produzidos dessa forma, de acordo com heurísticas decididas pelo usuário.

Contudo, uma solução como a proposta pelos autores tem algumas deficiências que a tornam insuficiente para o problema que queremos resolver no BCC. Em primeiro lugar, a gramática utilizada para representar os agrupamentos de recursos é incapaz de expressar certas restrições que estão presentes no BCC, como por exemplo a de não ter duas aulas de uma mesma matéria no mesmo dia, ou em dias subsequentes. Além disso, preferências dos professores não podem ser expressas, e como o problema prático resolvido trata de uma escola de ensino médio, os alunos não escolhem, na vasta maioria dos casos, de quais cursos participarão em um dado semestre. Concluímos que uma solução nesse estilo seria insuficiente para resolver o problema de construir uma grade com a mínima quantidade de conflitos para os alunos e as restrições presentes no BCC.

Outra proposta interessante foi desenvolvida por (Burke *et al.*, 1994), em que se trata o problema diretamente como um problema de colorir um grafo com uma certa quantidade (não conhecida a priori) de cores. A ideia geral é essencialmente a mesma que em (Cooper

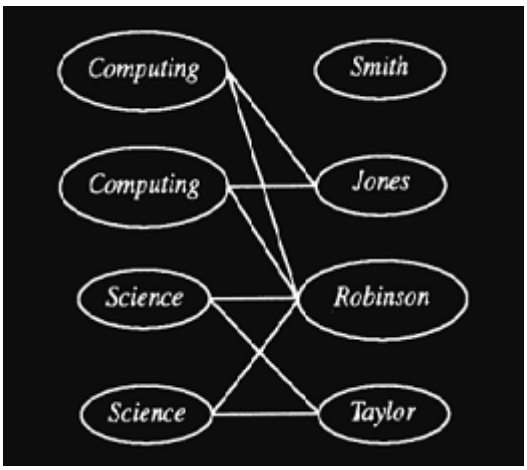


Figura 1.2: *Exemplo de um pareamento para um dado proto-tempo. Nesse caso, precisaríamos de 2 professores de computação e 2 professores de ciências. Não há pareamento possível, de modo que esse proto-tempo não pode ter as duas aulas de computação e ciências simultaneamente. Fonte: (Cooper e Kingston, 1993).*

e Kingston, 1993): é necessário encontrar conjuntos de disciplinas que podem ocorrer simultaneamente, isto é, que não competem por recursos. Constrói-se então um grafo de conflitos G, em que o conjunto de vértices V representa as disciplinas e existe uma aresta $(u, v), u \in V, v \in V$ se as disciplinas u e v compartilham algum recurso. Nesse caso, u e v não podem ocorrer simultaneamente. Por exemplo, em um dado horário, o professor A estará dando a aula 1 e portanto não pode dar a aula 2 nesse mesmo horário. Nesse caso, a aula 1 seria vizinha da aula 2 no grafo de conflitos. Ao colorir o grafo de modo que nenhuma aresta (u, v) seja tal que a cor de u é igual à cor de v, e associar cada tempo a uma cor, teremos também obtido uma partição das disciplinas tal que não há par de disciplinas que compartilham um recurso e ocorrem ao mesmo tempo.

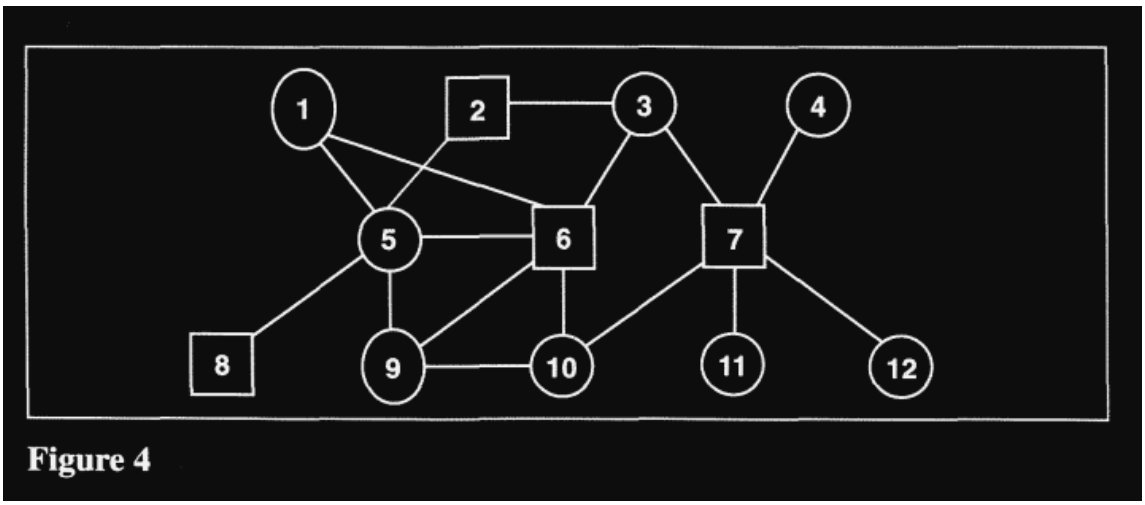


Figura 1.3: *Exemplo de coloração de um grafo para que não haja disciplinas simultâneas que usam o mesmo recurso. Nesse caso, a cor é representada pelo formato do vértice. Nesse grafo de conflitos, 3 horários são necessários. Fonte: (Burke et al., 1994).*

A vantagem do tratamento direto como um problema de coloração de um grafo é que

existem muitos algoritmos que são capazes de fornecer uma coloração razoável de um grafo em tempo polinomial (a coloração não será ótima, então o horário será construído com mais tempos do que o mínimo necessário, pois haverá mais cores do que o mínimo necessário). Caso o número de cores necessário seja maior do que o número de tempos disponível, será necessária intervenção do usuário para otimizar a solução produzida pelos algoritmos de coloração do grafo.

Para distribuir as turmas em salas, os autores usam duas versões de um algoritmo: Uma que permite a presença de duas turmas distintas na mesma sala (usado para construir horários de provas) e outra que não permite. Essencialmente a solução é a mesma: em ambos os casos, as turmas e salas são ordenadas em ordem crescente de tamanho necessário e tamanho disponível, respectivamente. O algoritmo procede para ocupar o máximo de capacidade da sala possível. Por exemplo, suponha que há uma sala S_1 de tamanho 30, uma sala S_2 de tamanho 60, e duas turmas: T_1 com tamanho 20, T_2 com tamanho 30 e T_3 com tamanho 40. Primeiro a turma T_1 é associada à sala S_1, pois o algoritmo avalia as turmas na ordem crescente. Em seguida descobre que a turma T_2 caberia na sala S_1 e ocuparia a sala mais completamente. Então a turma T_1 é movida para a sala S_2. a turma T_3 não cabe em S_1 mas cabe em S_2, compartilhando a sala com T_2. Veja a tabela abaixo para um exemplo do processo, em que cada coluna corresponde a uma iteração do algoritmo. O processo segue até que toda turma esteja distribuída ou que não haja mais salas disponíveis para deslocar uma turma (nesse caso mais períodos de tempo seriam necessários).

Capacidade	Iteração 1	Iteração 2	Iteração 3
`60`		$T_1(20)$	$T_1(20), T_3(40)$
`30`	$T_1(20)$	$T_2(30)$	$T_2(30)$

Tabela 1.1: *Método para distribuir turmas em salas, com compartilhamento*

O caso que não permite compartilhamento usa o mesmo processo, mas não haveria sala disponível para T_1 e um novo horário seria necessário. Notamos que a distribuição de turmas em salas com compartilhamento pode ser usado sem modificação para distribuir carga horária para professores. No caso em que não há professores suficientes, como não se pode contratar um professor adicional, a solução é quebrar uma matéria entre diversos professores ou aumentar a carga de aulas de algum professor.

Embora seja de arquitetura diferente, essa solução apresenta os mesmos problemas essenciais da solução proposta por (Cooper e Kingston, 1993): Não é possível modelar qualquer restrição, e as soluções do algoritmo são apenas aproximações que podem necessitar de ajuste fino e otimização por um usuário humano.

Alternativamente a métodos que envolvem modelar o problema como uma versão de algum problema que tem solução conhecida, é possível tratar diretamente o problema com a metodologia da pesquisa operacional, em que o conjunto de restrições é associado com uma série de custos para a violação de uma restrição e com o uso de uma função objetivo, que soma o valor de todos os custos e deve ser minimizada. Uma solução que usa programação inteira para modelar o problema pode ser encontrada em (Aubin e Ferland, 1989), em que um conflito (uso simultâneo de um recurso, seja classe, professor ou aluno) tem um custo associado por hora.

O modelo específico gerado pelos autores se baseia em dois conjuntos de variáveis de decisão. Considere primeiro que há N aulas e M tempos possíveis para o início de uma aula. Define-se, para cada aula i e tempo inicial j:

$$x_{ij} = \begin{cases} 1 & \text{, se a aula } i \text{ começa no tempo } j,\ 1 \leq i \leq N, 1 \leq j \leq M \\ 0 & \text{, caso contrário.} \end{cases} \tag{1.1}$$

Além disso, há S alunos e A seções (uma seção é o conjunto de tempos ocupados por uma aula). Define-se, para cada aluno s e seção a:

$$y_{sa} = \begin{cases} 1 & \text{, se o aluno } s \text{ tem aula na seção } a,\ 1 \leq s \leq S, 1 \leq a \leq A \\ 0 & \text{, caso contrário.} \end{cases} \tag{1.2}$$

Finalmente define-se uma função a ser otimizada e um conjunto de restrições:

$$\text{Min } f(x, y) = f_1(x) + \mu f_2(x, y)$$

(P) subject to

$$\sum_{j=1}^{m} x_{ij} = 1 \qquad 1 \leqslant i \leqslant n \tag{2.1}$$

$$x_{ij} = 0 \text{ or } 1 \qquad 1 \leqslant i \leqslant n,\ i \leqslant j \leqslant m \tag{2.2}$$

$$\sum_{a \in Q_w} y_{sa} = 1 \qquad 1 \leqslant s \leqslant S,\ w \in R_s \tag{2.3}$$

$$y_{sa} = 0 \text{ or } 1 \qquad 1 \leqslant s \leqslant S,\ 1 \leqslant a \leqslant A. \tag{2.4}$$

$$f_1(x) = f_{11}(x) + \rho f_{12}(x) + \nu f_{13}(x)$$

$$f_{11}(x) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$f_{12}(x) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{b=1}^{B} \sum_{t=1}^{T} \text{Max}\left\{0, \sum_{(i,j) \in K_{bt}} x_{ij} - U_b\right\}$$

$$f_{13}(x) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{n} \sum_{l=1}^{m} \delta_{ik} l_{ijlk} x_{ij} x_{kl}$$

Figura 1.4: *Modelo definido por (Aubin e Ferland, 1989), representando as restrições e função de custo $f = f_1(x) + \mu f_2(x, y)$ a ser otimizada. Fonte: (Aubin e Ferland, 1989) (editada).*

As restrições (2.1) e (2.2) representam, juntas, que toda aula i deve ter associado um horário de início j. Do mesmo modo, as restrições (2.3) e (2.4) significam que todo aluno é registrado em matérias para a qual se matriculou. O primeiro componente de f_1, f_{11}, representa o custo c_{ij} de iniciar a matéria i no horário j, de acordo com a preferência do professor responsável ($c_{ij} = 0$ se o professor achar o horário j ideal). Os valores de c_{ij} devem ser fornecidos por cada professor.

O segundo componente de f_1, f_{12}, representa o custo de exceder a quantidade de salas de um determinado tipo que estão disponíveis. T é o número de horas-aula durante a semana, B o número de tipos de salas, e U_b é a quantidade de salas do tipo b, $1 \leq b \leq B$, que existem. Finalmente, K_{bt} corresponde ao número de pares (i, j) de matérias i e horários j que ocupam uma sala do tipo b durante a hora t. De modo que, se houver 5 aulas que usam uma sala b em uma dada hora t, e houver 3 salas b, teremos $\sum_{(i,j)\in K_{bt}} x_{ij} - U_b = 5 - 3 = 2$.

O terceiro componente de f_1, f_{13}, é simples apesar de sua aparência: l_{ijkl} é a duração de um conflito entre a matéria i que começa em j e a matéria k que começa em l. $\delta_{ik} = 1$ se as matérias i e k compartilham algum recurso, como professor ou sala. Finalmente, os termos ρ, ν e μ são constantes definidas pelo usuário para dar peso relativo a essas 3 componentes. O último componente de f, $f_2(x, y)$, meramente conta quantas horas de conflito há para todos os alunos.

Para cada hora de conflito no horário gerado, o custo dos conflitos é contabilizado. Então, ocorrem trocas de horários entre disciplinas para minimizar os custos devido às salas e professores, e ocorrem trocas de estudantes entre grupos que assistem ao mesmo curso para diminuir o custo devido aos alunos. Para determinar essas trocas, o programa verifica se trocas simples de pares de disciplinas/alunos reduzirão o custo total. O método se repete até que não seja mais capaz de diminuir o valor da função de custo (encontre um mínimo local). O programa que soluciona esse modelo foi construído sob medida pelos autores, e usa um método heurístico para chegar a uma solução viável que tenha custo minimizado localmente (mas não necessariamente globalmente). Uma desvantagem significativa dessa solução é que uma solução inicial que seja viável (respeite todas as restrições rígidas) é necessária. Os autores se utilizam das tabelas horárias de anos passados como ponto de partida. Embora não seja um método ideal, o mesmo procedimento poderia ser adotado no BCC, já que tabelas de anos anteriores existem.

Outro exemplo de modelagem pode ser obtido em (Boland *et al.*, 2008). Nesse caso, os autores se valem de uma proposta mista, usando um conjunto de modelagem do problema como um problema de otimização e decomposição do problema em 2 subproblemas: O agrupamento de disciplinas em blocos que podem ser simultâneos e sua população com alunos, e a distribuição dos blocos na semana. Em seguida vários modelos distintos são propostos para o primeiro subproblema, e sua solução obtida com pacotes prontos de otimização inteira. O modelo produzido, que conta com mais de 20 restrições diferentes, e 3 subconjuntos específicos de restrições para relaxar diferentes aspectos do problema, ilustra como a complexidade desse tipo de solução escala rapidamente, quando queremos aumentar o poder de expressão do modelo.

Uma desvantagem dos métodos de pesquisa operacional que pode ser deduzido da leitura dos artigos citados é que os modelos construídos podem mudar significativamente com a adição de restrições particulares de cada instituição, e que a construção de modelos corretos não é trivial nem geral para todos os problemas que podem ser encontrados. Embora os métodos tenham a vantagem de conseguir tratar qualquer tipo de restrição, a expressão das restrições muitas vezes pode ser muito difícil, e o modelo gerado pode tornar necessária a produção adicional de um algoritmo para resolvê-lo, uma vez que não há garantia que algoritmos conhecidos serão aplicáveis ou retornarão uma resposta aceitável em tempo razoável.

Outra proposta interessante é o uso de *Answer Set Programming* (ASP), um paradigma de programação declarativa para problemas de busca com sintaxe similar à `prolog`. Uma descrição breve de ASP pode ser encontrada em (Lifschitz, 2008). Como exemplo de sintaxe, considere a regra ASP p `:-` q. A resolução de um programa ASP que só contém essa regra será p, q. Isso representa que, se perguntássemos ao programa se temos os átomos p, q em conjunto, a resposta é sim - pois $p \implies q$. Combinando p `:-` q com $\{s, t\}$ `:-` p, indicamos que s ou t podem gerar p. Os conjuntos de átomos válidos serão

$$p, q, s$$

$$p, q, t$$

$$p, q$$

$$s, t, p, q$$

Há também regras restritivas que são representadas por implicações sem o lado esquerdo, como por exemplo

$$\texttt{:-}\ s, \texttt{not}\ t$$

significa que um modelo estável não pode ser gerado com o átomo s e sem o átomo t - não há nada que implica esse modelo. adicionar essa regra ao par de regras acima exclui a resposta p, q, s. Além disso há regras numéricas, por exemplo $1\{p, q, s\}2$ significa escolha pelo menos 1 e no máximo 2 átomos do conjunto $\{p, q, s\}$. Letras maiúsculas indicam variáveis. por exemplo, $p(a), p(b), p(c)$ e $p(X)$ `:-` $q(X), X \neq a$ resulta no modelo estável $p(a), p(b), p(c), q(b), q(c)$.

Há algumas vantagens de se construir uma solução para o problema de construção de horários usando esse tipo de linguagem: Em primeiro lugar, é possível expressar muitas das restrições que os métodos de transformação em outros problemas são incapazes de expressar, por meio da definição de regras que introduzem uma penalização numérica na solução, além de regras que obrigatoriamente devem ser cumpridas. Dessa forma, preferências particulares da instituição, como por exemplo algum espaçamento particular mínimo de aulas na semana, são relativamente fáceis de expressar, embora a sintaxe não seja imediatamente óbvia para um leigo.

Outra vantagem do paradigma ASP é que já existem algoritmos bem estabelecidos que encontram respostas para instâncias em tempo razoável (embora o problema arbitrário de encontrar satisfatibilidade seja também NP-difícil). Portanto, uma solução para uma dada instância do problema via ASP é limitada principalmente pela capacidade do operador do sistema de expressar as restrições presentes na sintaxe do programa, algo que pode ser facilmente feito por um usuário especialista que tenha experiência com programação, que seria o caso dos professores do BCC. Um exemplo de solução do problema nessas linhas pode ser encontrado em (Banbara *et al.*, 2013).

O modelo definido pelos autores é composto por 4 restrições rígidas, que sempre devem ser obedecidas, e por 9 restrições não-rígidas que podem ser quebradas, mas que incorrem uma penalidade (um custo) ao ser quebradas. As restrições rígidas utilizadas podem ser examinadas na figura abaixo.

```
% H1. Lectures
N { assigned(C,D,P) : d(D) : ppd(P) } N :- course(C,_,N,_,_,_).

% H2. Conflicts
:- not { assigned(C,D,P) : course(C,T,_,_,_,_) } 1, t(T), d(D), ppd(P).
:- not { assigned(C,D,P) : curricula(Cu,C) } 1, cu(Cu), d(D), ppd(P).

% H3. RoomOccupancy
1 { assigned(C,R,D,P) : r(R) } 1 :- assigned(C,D,P).
:- not { assigned(C,R,D,P) : c(C) } 1, r(R), d(D), ppd(P).

% H4. Availability
:- assigned(C,D,P), unavailability_constraint(C,D,P).
```

Figura 1.5: *Restrições rígidas do modelo ASP definido por (Banbara et al., 2013). Fonte: (Banbara et al., 2013).*

Para a restrição H_1, a regra garante que para todo curso `course(C, _, N, _, _, _)` com N aulas, valham exatamente N predicados `assigned(C, D, P)` para algum dia D e período P.

Para a restrição H_2, a primeira regra garante que para todo professor T, dia D e período P, há no máximo 1 curso que T ensina para o qual vale o predicado `assigned(C, D, P)` (A sintaxe `not {p, q}1` significa nenhum de p, q ou um dos elementos). A segunda regra é essencialmente a mesma, mas garante que para todo currículo Cu o dia D, período P e curso C há no máximo um curso C que pertence a Cu tal que vale o predicado `assigned(C, D, P)`. Essas regras garantem que não há conflitos para professores nem de disciplinas obrigatórias para os alunos.

Para a restrição H_3, a primeira regra garante que para todo predicado `assigned(C, D, P)` há exatamente uma sala R tal que vale `assigned(C, R, D, P)`. A segunda regra garante que para toda sala R, dia D e período P há no máximo um curso C tal que vale o predicado `assigned(C, R, D, P)`. Isso garante que toda aula tem uma sala e nenhuma sala tem duas aulas simultâneas.

Para a restrição H_4, a regra garante que não haverá nenhum elemento para o qual valem os dois predicados, já que se o curso C não está disponível no período P do dia D, ele não pode estar associado a esse período.

As restrições não rígidas podem ser violadas e sua violação está associada a um valor. Descreveremos como enunciar 3 das restrições rígidas, que podem ser examinadas abaixo.

```
% S1. RoomCapacity
penalty("RoomCapacity",assigned(C,R,D,P),(N-Cap)*penalty_of_room_capacity) :-
        assigned(C,R,D,P), course(C,_,_,_,N,_), room(R,Cap,_), N > Cap.

% S2. MinWorkingDays
working_day(C,D) :- assigned(C,D,P).
penalty("MinWorkingDays",course(C,MWD,N),(MWD-N)*penalty_of_min_working_days) :-
        course(C,_,_,MWD,_,_), N = [ working_day(C,_) ], N < MWD.

% S3. IsolatedLectures
scheduled_curricula(Cu,D,P) :- assigned(C,D,P), curricula(Cu,C).
penalty("IsolatedLectures",isolated_lectures(Cu,D,P),penalty_of_isolated_lectures) :-
        scheduled_curricula(Cu,D,P), not scheduled_curricula(Cu,D,P-1),
        not scheduled_curricula(Cu,D,P+1).
```

Figura 1.6: *3 Restrições não-rígidas do modelo ASP definido por (Banbara et al., 2013). Fonte: (Banbara et al., 2013).*

Sobre a restrição S_1, para cada sala R com capacidade Cap tal que o número de alunos N no curso C associado à sala por `assigned(C, R, D, P`, cria um átomo de penalidade `penalty("RoomCapacity", assigned(C, R, D, P), (N - Cap)*penalty_of_room_capacity)`, isto é, uma penalidade que equivale a um múltiplo da penalidade definida pelo usuário por exceder a capacidade da sala.

Sobre a restrição S_2, para cada predicado `assigned(C, D, P)` é gerado um átomo `working_day(C, D)` e então para cada curso tal que suas aulas devem ser espalhadas em um mínimo de dias diferentes igual a MWD, soma os dias em que o curso ocorre com N = `[ working_day(C, _) ]` e aplica um átomo de penalidade `penalty("MinWorkingDays", course(C, MWD, N), (MWD - N)*penalty_of_min_working_days)`. Novamente, a penalidade é uma das constantes definidas pelo usuário.

Sobre a restrição S_3, para cada currículo Cu com o curso C, e cada predicado `assigned(C,D,P)`, criamos um predicado que indica que o currículo Cu tem um compromisso no dia D, período P. Em seguida, criamos uma penalidade para todo compromisso que não tem um compromisso no período P+1 e P-1, no mesmo dia. Novamente a penalidade para cada ocorrência deve ser definida pelo usuário.

Finalmente se utiliza uma função de otimização para definir que a soma do custo de todas as penalidades deve ser mínimo:

```
#minimize [ penalty(_,_,P) = P ].
```

Figura 1.7: *Função de minimização de custo modelo ASP definido por (Banbara et al., 2013). O otimizador #minimize significa que deve ser escolhido o modelo estável com soma mínima do valor P associado aos átomos de penalidade. Fonte: (Banbara et al., 2013).*

Os autores obtiveram melhoras em diversos *benchmarks*, embora para alguns problemas a solução seja pior que o *benchmark*. Uma outra desvantagem observada é que o tempo tomado pelo *ASP-solver* e as soluções obtidas podem ser dependentes de duas formulações

diferentes das mesmas restrições. Essa desvantagem se relaciona com a mencionada anteriormente para a metodologia da pesquisa operacional: A definição de modelos diferentes para o mesmo problema tem impacto na tratabilidade do problema. Entretanto, os modelos em ASP são consideravelmente mais simples do que os obtidos por métodos de pesquisa operacional.

Em conclusão, podemos afirmar que o melhor método para resolver esse tipo de problema parece ser algum tipo de modelagem, pois apenas modelagem é capaz de expressar todos os tipos de restrições que poderiam surgir, por meio de métodos como a introdução de custos e a definição de restrições obrigatórias, embora a transformação do problema em outros problemas pode ser útil em casos particulares. Dos métodos de modelagem comuns, o uso de paradigmas declarativos como o ASP parece ser superior à modelagem como problemas de otimização em razão da maior simplicidade dos modelos ASP, do seu poder de expressão, e da disponibilidade de solucionadores automáticos sem a necessidade potencial de desenvolver um algoritmo para resolver um modelo. Em relação aos métodos que envolvem redução a algum outro problema, embora esses métodos possam ser mais simples de implementar ou mais rápidos, eles não são capazes de capturar todos os aspectos do problema e portanto não servem como solução geral.

Capítulo 2

Resultados e Discussão

2.1 Arquitetura da Solução

Discutiremos brevemente quais são os aspectos da solução implementada, que se divide em 3 seções principais: o formulário de intenção de matrícula, o programa que modela os cursos e os alunos, e um terminal que permite realizar consultas à base de dados produzida no programa.

2.1.1 Formulário de Matrícula

A solução construída tem 3 estágios. Em primeiro lugar, publica-se um formulário usando a ferramenta Google Forms (*Google Forms* 2023), que permite a construção de formulários de forma simples e rápida, e apresenta ferramentas de autenticação suficientes para garantir que apenas e-mails de um certo domínio respondam, e que cada e-mail responda apenas uma vez, características necessárias para garantir a acurácia da pesquisa. A criação de um formulário para um semestre típico leva em média 1h, e a resposta de um usuário pode ser preenchida em menos de 3 minutos. Um exemplo de formulário típico pode ser observado na figura 2.1.

Uma vez que o autor do formulário decida encerrar a pesquisa, é possível baixar as respostas obtidas até o momento em formato .csv, que pode ser fornecido como entrada para o programa desenvolvido. Nesse momento o usuário tem na base de dados um retrato das intenções de matrícula dos respondentes da pesquisa.

Cabe notar que o engajamento na resposta a essa pesquisa é um ponto de falha importante da solução. No teste piloto realizado, apenas 40 respostas foram obtidas de um total aproximado de 200 alunos matriculados. Embora o teste tenha sido divulgado de forma limitada, por meio de distribuição de um link para o formulário no aplicativo de mensagens Telegram, melhorias no engajamento de resposta dos alunos serão imperativas para o uso eficaz da solução. Discutiremos posteriormente como aumentar o engajamento.

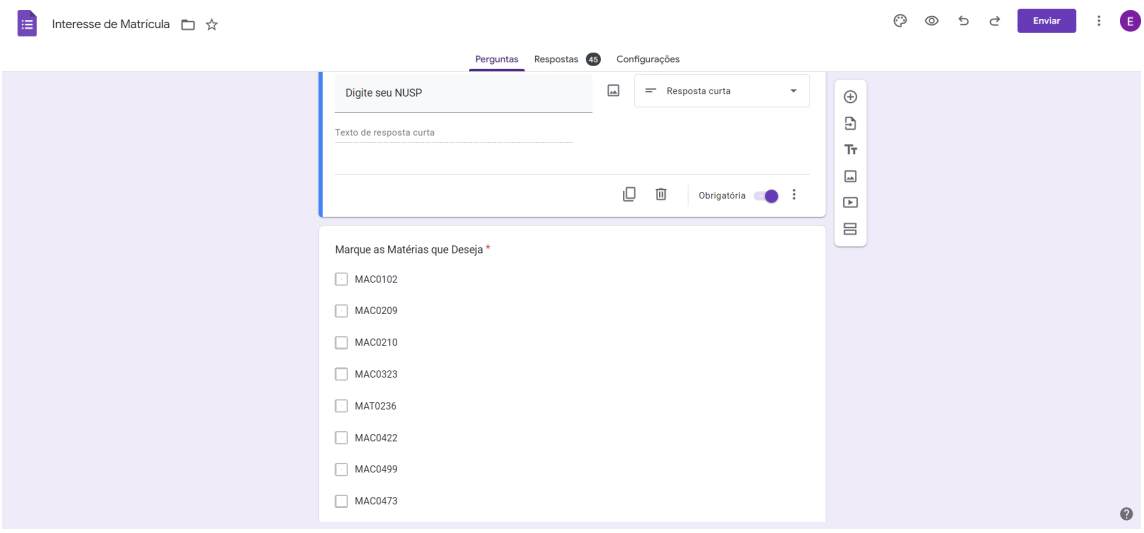


Figura 2.1: *Exemplo de um forms típico.*

2.1.2 Administrador de Conflitos

O segundo objeto que compõe a solução é um programa em C++ que denominamos Administrador de Horários. O código produzido está disponível no GitHub em (*Administrador de Conflitos* 2023). Em primeiro lugar é necessário carregar no programa dois conjuntos de dados: As informações sobre as intenções de matrícula dos alunos e a grade horária em que as disciplinas estão organizadas. As informações de matrícula podem ser obtidas diretamente do Forms e são lidas automaticamente pelo programa, bastando passar o arquivo na linha de comando. As informações sobre a grade horária, contudo, não podem ser obtidas facilmente pois são publicadas em uma planilha pelos RDs. Desse modo o RD precisa compilar um pequeno arquivo de texto que relaciona as siglas das disciplinas com seus horários. Um exemplo de arquivo no formato adequado será fornecido junto com o programa.

Uma vez que os dados necessários são carregados no programa, são separados em 3 tipos de dados: Course, Student e TimeWindow. Esses 3 tipos de dados representam uma disciplina, um aluno, e um espaço de tempo na semana, que é definido por um dia da semana e um horário de início e término. Com o uso dos dois arquivos de entrada, os estudantes são alocados em Students que se matricularão em um conjunto de Courses, e cada Course tem um conjunto de TimeWindow que representa seus horários na semana. Esses 3 objetos são então unidos em uma relação denominada TimeChart, que contém *handles* para todos os dados e é capaz de determinar a quantidade de conflitos. Foi utilizada uma arquitetura orientada a objetos e estruturas de dados implementadas na biblioteca padrão do C++. Seguiu-se um padrão de *design bottom-up*, em que os objetos foram implementados primeiro e depois integrados na relação TimeChart. O diagrama de relacionamento abaixo ilustra as relações entre os objetos. Discutiremos as características dos objetos implementados.

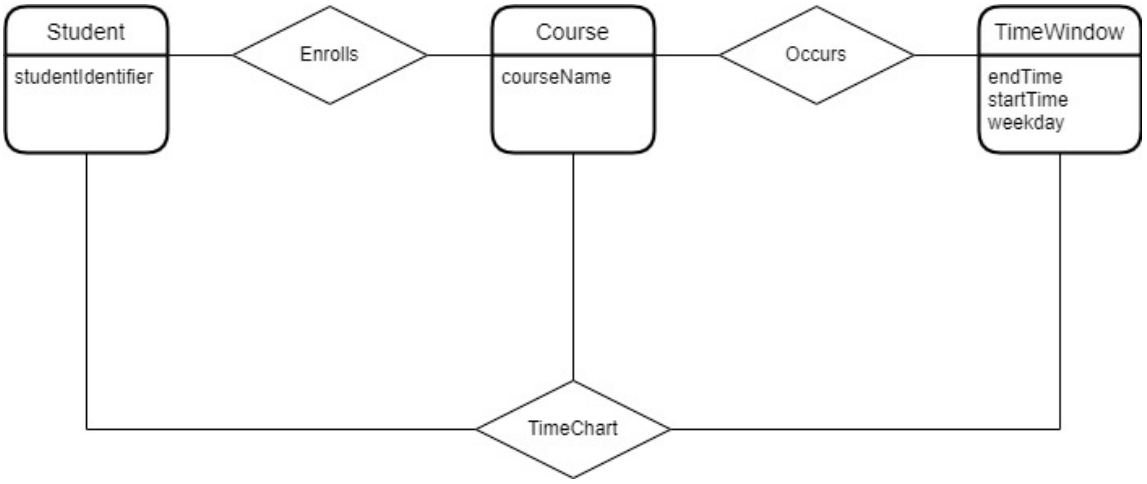


Figura 2.2: *Diagrama de relacionamento entre os objetos implementados.*

Course

Um Course é um objeto que representa um curso, uma disciplina no BCC. Um curso contém os seguintes campos de dados:

```
ll id;
std::string name;
std::set<TimeWindow, TWComparator> time_windows;
static ll count;
```

Em que `id` é um identificador único para cada curso, `name` é o nome do curso, e o conjunto `time_windows` representa todos os horários usados por aquele `Course`. `count` serve para gerar as identidades e é uma variável de classe.

Os métodos implementados em `Course` são:

```
Course(std::set<TimeWindow, TWComparator> time_windows, std::string name);
Course(const Course& other);
~Course();
std::set<TimeWindow, TWComparator> getTimes();
std::string getName() const;
ll generateId();
ll getId() const;
void addTimeWindow(TimeWindow time_window);
void removeTimeWindow(TimeWindow time_window);
std::string to_string();
```

O significado e função de vários desses métodos como *getters* e *setters* é o esperado pelos seus nomes. A necessidade de implementação de construtores de cópia e de um destrutor específico é uma regra geral em C++ discutida mais adiante. Os métodos que recebem `TimeWindow` como parâmetro operam sobre o conjunto de horários da disciplina. Finalmente, é necessário implementar um comparador:

```
struct CourseComparator{
    bool operator () (const Course left, const Course right) const{
        return left.getId() < right.getId();
    }
};
```

Um comparador deve ser fornecido a todo `set` na biblioteca padrão de C++ com o propósito de definir se dois objetos são iguais (pois um `set` contém elementos únicos, não pode ocorrer duplicação de elementos em um conjunto). Se um comparador não for implementado, o comparador padrão será usado, e ele terá comportamento indefinido se usado em objetos definidos pelo usuário ou ponteiros.

Note que o comparador implementado é dependente direto do fato de que todo objeto tem Id único. Isso é o motivo da necessidade de implementar um construtor de cópia e um destrutor, ao invés de utilizar os padrões da linguagem.

Student

Um objeto `Student` representa um aluno matriculado no BCC. Suas variáveis são:

```
ll id;
std::set<Course, CourseComparator> desired_courses;
ll NUSP;
static ll count;
```

O identificador único do estudante é o campo id. Cada aluno contém um `set` de `Course` que representa as matérias em que pretende se matricular. A associação do conjunto de matérias do aluno com os horários de cada matéria permite determinar o número de conflitos.

Os métodos implementados em `Student` são:

```
Student(std::set<Course, CourseComparator> desired_courses);
Student(std::set<Course, CourseComparator> desired_courses, ll NUSP);
Student(const Student& other);
~Student();
std::set<Course, CourseComparator> getCourses() const;
ll getId() const;
ll getNUSP() const;
std::string to_string() const;
```

Os métodos em `Student` são exatamente análogos aos métodos já discutidos em `Course`. O construtor adicional `Student(std::set<Course, CourseComparator> desired_courses);` que não inicializa `NUSP` existe apenas para propósitos de testagem. O comparador implementado para `Student` é imediato:

```
struct StudentComparator{
    bool operator()(const Student a, const Student b) const{
        return a.getId() < b.getId();
    }
};
```

TimeWindow

Um objeto `TimeWindow` representa uma janela de tempo, isto é, um período de tempo em um dia da semana. Suas variáveis são:

```
int start_hour;
int end_hour;
int week_day;
ll id;
static ll count;
```

`start_hour` indica a hora em que a janela se inicia e `end_hour` a hora em que termina. Por exemplo, um curso que começa às 14:00 e dura até às 16:00 teria `start_hour` = 14 e `end_hour` = 16. `week_day` representa o dia da semana e é convertido entre texto e string por um par de enumeradores:

```
std::map< std::string, int > Weekday{
    {"monday", 0}, {"tuesday", 1}, {"wednesday", 2}, {"thursday", 3},
    {"friday", 4}, {"saturday", 5}, {"sunday", 6}
};

std::map< int, std::string > invWeekday{
    {0, "monday"}, {1, "tuesday"}, {2, "wednesday"}, {3, "thursday"},
    {4, "friday"}, {5, "saturday"}, {6, "sunday"}
};
```

Permitindo que o usuário escreva em texto puro os dias da semana e que o programa os utilize como índices de vetores. O horário semanal foi modelado como uma matriz `chart[7][24]` representando as 24 horas dos 7 dias da semana no objeto `TimeChart`.

O comparador implementado é o esperado:

```
struct TWComparator {
    bool operator() (const TimeWindow lhs, const TimeWindow rhs) const {
        return lhs.getId() != rhs.getId();
    }
};
```

Os métodos de `TimeWindow` são:

```
TimeWindow(int start_hour, int end_hour, std::string weekday);
TimeWindow(const TimeWindow& other);
TimeWindow();
~TimeWindow();
int getStartingHour() const;
ll getId() const;
int getEndingHour() const;
int getWeekday() const;
int getDuration() const;
std::string to_string() const;
bool clashes(TimeWindow other) const;
```

```
    bool equalTo(TimeWindow other) const;
};
```

Em que o construtor vazio existe apenas para propósitos de testagem. Os *getters* fazem o que se espera, retornam a variável nomeada, com a exceção de `getDuration`, que retorna `end_hour - start_hour`. O método `equalTo` define se duas janelas de tempo tem o mesmo dia da semana, o mesmo horário de início e o mesmo horário final (mas não compara as identidades). O método `clashes` define se um conflito ocorre. Seu funcionamento é:

```
bool TimeWindow::clashes(TimeWindow other) const{

    if (this->getWeekday() != other.getWeekday()){
        return false;
    }
    if (this->getEndingHour() <= other.getStartingHour() ||
        this->getStartingHour() >= other.getEndingHour()){
        return false;
    }
    return true;
}
```

Um conflito foi definido de forma simples, a saber: Quando duas disciplinas tem horários intercalados, há um conflito. Um horário intercalado é um par de TimeWindow que ocorrem no mesmo dia da semana, e tal que o horário final da primeira ocorre depois do horário inicial da segunda, ou que o horário inicial da segunda ocorre antes do final da primeira. Pelo código, pode-se ver que se o horário de término da primeira for exatamente igual ao horário de início da segunda, ou o horário de início da segunda for exatamente igual ao horário de término da primeira, não há conflito. A figura abaixo ilustra o processo.

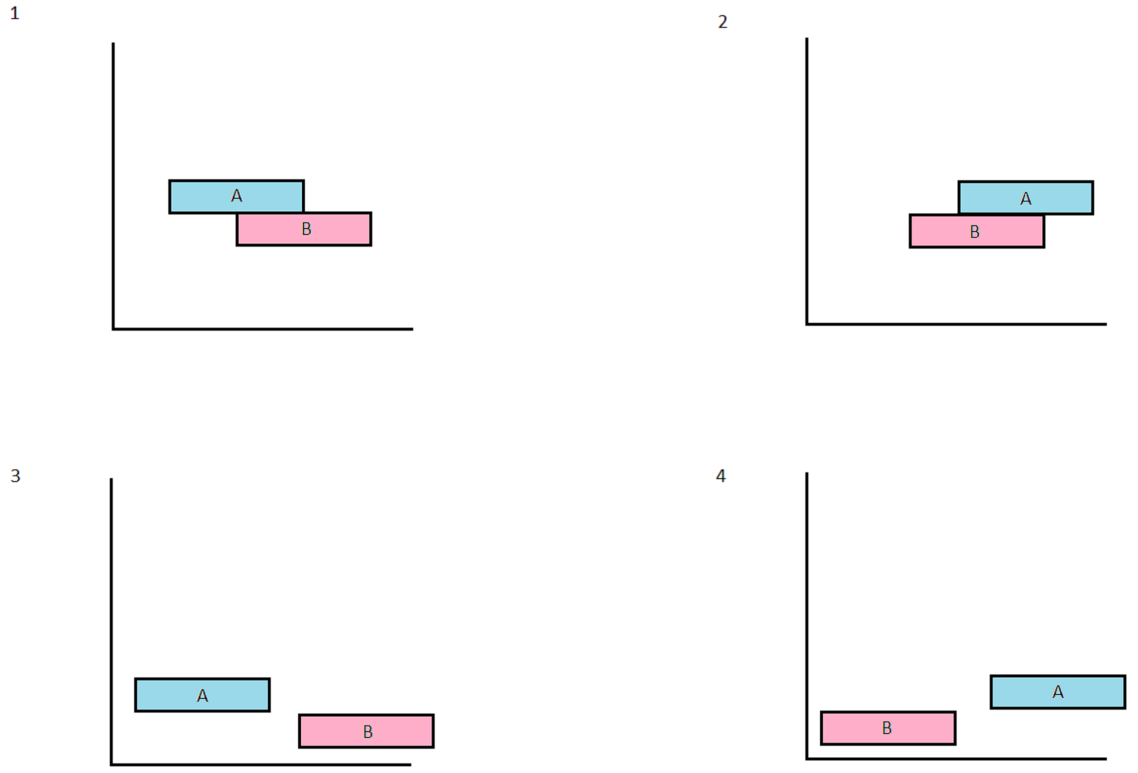


Figura 2.3: *Combinações de horários possíveis. No quadro 1, A termina depois que B começa, e temos um conflito. No quadro 2, B termina depois que A começa, e temos um conflito. Nos casos contrários, não há conflito.*

TimeChart

`TimeChart` é o objeto que integra os 3 objetos descritos acima e o principal objeto do programa. Cada instância do programa produz exatamente uma instância do objeto `TimeChart`. Suas variáveis são:

```
std::set<Course, CourseComparator> chart[7][24];
std::set<Student, StudentComparator> students;
```

O `set` de `Student` representa todos os alunos matriculados que foram inseridos como dados no programa (os respondentes do Forms). Já a matriz `chart[7][24]`, como já mencionado, representa todas as horas disponíveis em uma semana. Ela é uma matriz de `set` de `Course`: Cada um desses `set` representa os cursos que ocorrem no dia *i*, hora *j* da matriz. Por exemplo, se MAC0110 ocorre às segundas, das 8 às 10, temos que o `set` dado por `chart[0][8]` contém MAC0110. O mesmo vale para `chart[0][9]`.

Os métodos de `TimeChart` são:

```
TimeChart(std::set<Student, StudentComparator> students);
TimeChart();
void setCourseAtWindow(Course& course, TimeWindow time_window);
void removeCourseAtWindow(Course course, TimeWindow time_window);
void addCourse(Course& course);
```

```
void removeCourse(Course course);
void addStudent(Student& student);
void removeStudent(Student student);
std::set<Course, CourseComparator> getCoursesAtWindow(TimeWindow time_window);
std::set<Student, StudentComparator> getStudents();
void swapCourses(Course first_course, TimeWindow first_window,
Course second_course, TimeWindow second_window);
int findConflicts();
void printChart();
```

Como o objeto é único, não há necessidade de um construtor de cópia e destrutor para administrar identidades únicas. Todos os métodos exceto `findConflicts` e `printChart` manipulam os conjuntos de alunos, cursos e horários. o método `swapCourses` troca dois cursos de lugar na grade, de acordo com as posições `first_window` e `second_window`. O método `findConflicts` é o principal responsável por obter conflitos.

Com a definição de conflitos usada, para determinar o número de conflitos em uma grade horária, o programa passa por todos os alunos cadastrados e avalia qual é o seu conjunto de cursos. Para cada par de cursos no conjunto, as janelas de tempo são avaliadas: para cada par de janelas, verifica-se se há um conflito. A quantidade total de conflitos é contabilizada e pode ser informada ao usuário. Desse modo há uma métrica capaz de quantificar o efeito de uma determinada mudança de horário sobre a quantidade global de conflitos.

Terminal

O módulo Terminal corresponde a uma pequena biblioteca de manipulação de strings que foi implementada para certas funcionalidades desejadas para interação. A interação com o usuário foi implementada no módulo principal (main) do módulo de testes. Desse modo, quando o programa é executado, ele roda os testes e em seguida permite a interação com o usuário. Futuramente temos a intenção de separar os dois módulos e chamar os testes por meio de uma função.

2.1.3 Terminal de Interação

O elemento final da solução é o módulo que permite ao usuário interagir com os dados e formular pesquisas sobre potenciais alterações de horários. Atualmente, o método selecionado para permitir a interação do usuário e viabilizar os testes piloto foi um terminal em texto, por meio do qual o usuário digita comandos e seus argumentos. Uma série de comandos foi implementada e é relacionada a seguir. Nos comandos, um argumento entre colchetes [argumento] é obrigatório, e entre parênteses (argumento) é opcional.

- show [courses | students] imprime na tela o conjunto de cursos ou alunos que estão cadastrados no programa.

- add course [coursename] [wday] [stT] [enT] cadastra o curso de nome coursename e janelas de tempo definidas pelos outros argumentos. A quantidade de janelas pode ter qualquer tamanho.

- add student [course] cadastra um aluno com um conjunto de cursos definido. A lista de cursos pode ter qualquer tamanho.

- remove course [identifier] remove o curso com o identificador dado do conjunto de cursos.

- remove student [identifier] remove o aluno com o identificador dado do conjunto de alunos.

- set course [cId] [wDay] [stT] [enT] adiciona uma janela de tempo com os 3 parâmetros finais ao curso identificado por cId.

- unregister time [cId] [wDay] [stT] [enT] remove uma janela de tempo com os 3 parâmetros finais do curso identificado por cId.

- conflicts lista a quantidade de conflitos presente no TimeChart atual.

- save [filename] salva o TimeChart atual em um .csv

- load [filename] carrega um TimeChart a partir de um .csv

Os comandos implementados permitem ao usuário a interação necessária para criar alterações numa grade e avaliar seu efeito. Originalmente tínhamos a intenção de implementar uma interface que permitisse várias funções relevantes de interação, como uma visualização gráfica de uma grade de horários, menus laterais que permitissem a visualização e edição de disciplinas e seus horários, e funcionalidade que permitisse arrastar determinados horários na grade por meio de cliques com um *mouse*. Além disso seria útil implementar funções que permitissem ao usuário cancelar uma mudança recém-feita, por meio da implementação de uma fila de estados que pudesse ser recuperada. Uma implementação dessas funções de interação naturalmente usará as funções que já estão implementadas no terminal, bastando traduzir os *inputs* na interface gráfica para argumentos dos métodos implementados. Adicionamos a seguir um esboço da interface sugerida, que não pôde ser implementada em tempo.

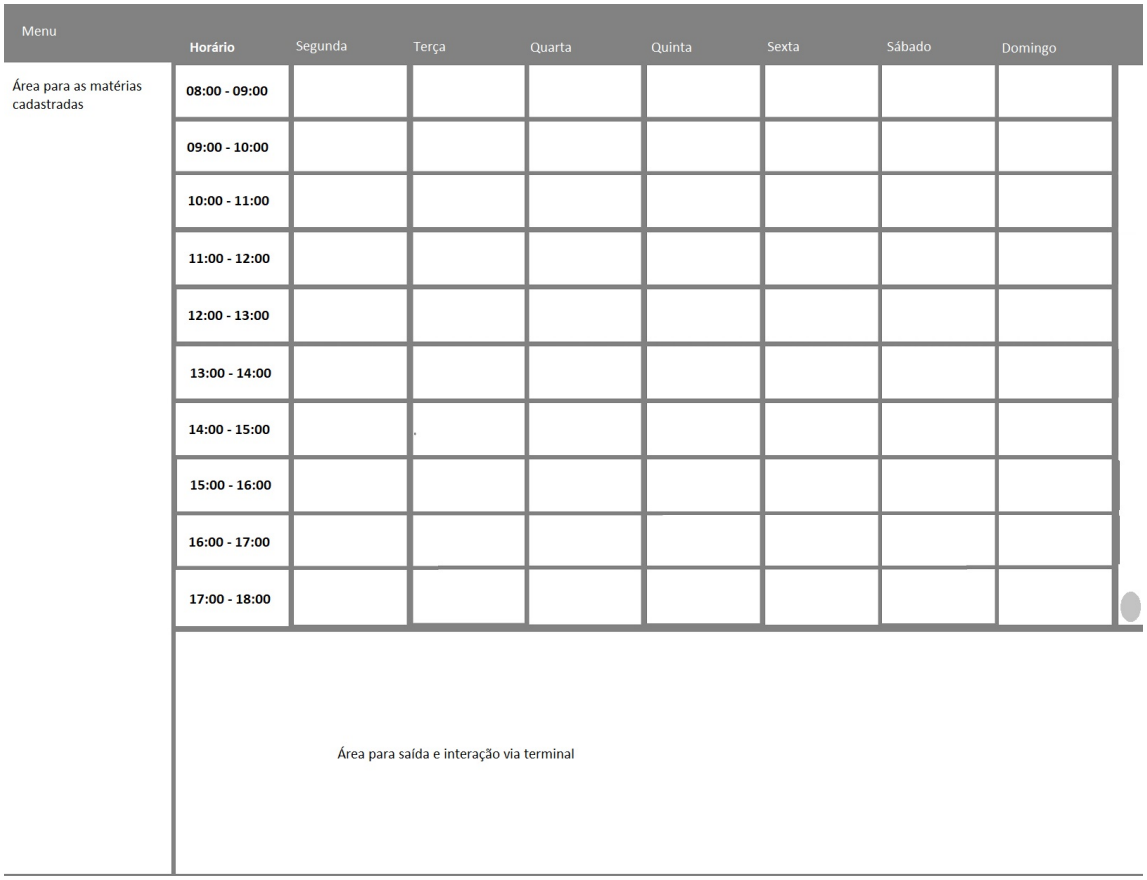


Figura 2.4: *Exemplo da interface desejada: essencialmente, uma representação visual da variável* `chart` *no objeto* `TimeChart`.

Embora o terminal como implementado não seja ideal para um usuário leigo, é suficiente para um usuário técnico como um RD utilizar a solução. Os elementos necessários para testar a solução com um teste piloto foram implementados.

2.2 Testes

Para realizar os testes, foi criado um *toy dataset* que fosse representativo do possível horário de um semestre, e alunos foram cadastrados em disciplinas nesses horários. Foram realizados dois conjuntos de testes, os testes de unidade com o *toy dataset* e o teste piloto com dados reais.

2.2.1 Testes de Unidade

Para verificar a corretude do código implementado, foram escritos uma série de testes de unidade sobre alunos do *toy dataset* que determinam se nossa implementação dos objetos funciona corretamente. Testamos se os comparadores funcionam como esperado, se o gerador de identidades únicas funciona corretamente, e todos os casos possíveis de conflitos, para verificar se os conflitos são contados corretamente pelo programa. O uso de

testes de unidade consistentes permite garantir que certas funcionalidades necessárias são preservadas mesmo com mudanças no funcionamento interno dos objetos.

2.2.2 Teste Piloto

O teste piloto foi um teste que pretendeu simular condições reais de funcionamento do software. Para realizá-lo, portanto, foi necessário seguir a arquitetura da solução como se fôssemos um RD realizando análises para o horário de um determinado semestre. Em primeiro lugar foi criado um Forms com as disciplinas que serão oferecidas no 1º semestre de 2023. Esse forms foi distribuído aos alunos do BCC via grupo de mensagens no aplicativo Telegram. O período de coleta das respostas foi de 1 semana. Ao fim de 1 semana, havia 45 respostas ao formulário, de um total estimado de aproximadamente 250 alunos. Em seguida foi criada a tabela de horários para entrada no programa que foi formulada a partir do horário estabelecido pela comissão do BCC. Os dados foram cadastrados no Administrador de Horários e análises sobre o total de conflitos foram realizadas, considerando os pedidos de mudança reais que foram encaminhados ao RD nesse semestre. O principal pedido de mudança foi que a disciplina de Inteligência Artificial (MAC0425) não conflitasse com a disciplina de Introdução ao Desenvolvimento de Sistemas de Software (MAC0350). Foi sugerido pelos alunos que MAC0350 mudasse seus horários de terça às 10h e quinta às 8h para terça às 8h e sexta às 8h (Sugestão 1), ou quarta às 10h e sexta às 8h (Sugestão 2). Para avaliar o pedido, realizamos simulações de horário com ambas as alterações e contabilizamos os conflitos.

Original	Sugestão 1	Sugestão 2
32	31	31

Tabela 2.1: *Quantidade total de conflitos em cada um dos horários sugeridos*

No horário original há 6 conflitos entre MAC0425 e MAC0350, e 2 conflitos entre MAC0350 e Algoritmos e Estruturas de Dados II (MAC0323). Na sugestão 1 há 4 conflitos entre MAC0350 e Introdução ao Aprendizado de Máquina (MAC0460) e 3 conflitos entre MAC0350 e Laboratório de Métodos Numéricos (MAC0210). Há 8 alunos pedindo essa mudança na planilha, de modo que os números do teste piloto são muito próximos aos da planilha. Notamos que a mudança sugerida geraria a melhoria mínima possível na quantidade de conflitos, e por isso, embora tecnicamente devesse ser feita, provavelmente não valeria a pena levar essa sugestão ao corpo docente. Na sugestão 2 há 6 conflitos entre MAC0350 e Programação Concorrente e Paralela (MAC0219) e 1 conflito entre MAC0350 e Noções de Probabilidade e Processos Estocásticos (MAE0228). Logo valem as mesmas ponderações que para a sugestão 1. A análise das 3 sugestões levou menos de 30 minutos para ser feita, considerando as alterações nos horários feitas pelo usuário, e a avaliação das respostas do programa para identificar onde estavam os conflitos em cada horário.

2.3 Discussão

Notamos que embora o programa tenha possibilitado avaliar o valor relativo das sugestões dos alunos no nosso teste piloto, há alguns problemas a ser solucionados. O principal problema é obter uma resposta mais expressiva à pesquisa. Consideramos que para que os resultados sejam de fato representativos, uma proporção significativamente maior de alunos deve ter respondido ao Forms. É claro que, em semestres futuros, poderemos disponibilizar a pesquisa de modo a ter mais tempo para obter respostas, e poderemos divulgar mais intensamente, inclusive com visitas às salas de aula, que não foi feito para o teste piloto. Uma estratégia que deve ser usada também é a de divulgar a pesquisa antes da publicação do forms, já que (Sammut *et al.*, 2021) indicam que a pré-notificação dos respondentes melhora a participação nesse tipo de pesquisa. Outra estratégia citada pelos autores para melhorar a taxa de resposta e que pode ser implementada é a explicação do propósito da pesquisa e como a resposta beneficiará o respondente.

Outra informação que obtivemos do teste piloto foi a necessidade de implementar algumas *features* de consulta que permitam ao usuário determinar quais são os conflitos de uma disciplina específica, ao invés de simplesmente obter a grade total de conflitos. Isso permitirá ao usuário saber exatamente quais pares de disciplinas estão causando os conflitos e auxiliará ao decidir quais disciplinas podem ser movidas. Isso é de especial relevância quando queremos que disciplinas de certos períodos não conflitem, por exemplo. Essa alteração também tornaria a tarefa de avaliar o valor relativo de uma sugestão mais rápida.

Consideramos que a solução obtida, apesar de suas limitações, será de grande utilidade para o RD se houver resposta expressiva à pesquisa, pois é possível agora saber de antemão o valor relativo de uma dada proposta. Com os métodos anteriores, se o RD decidisse acatar a sugestão de mudança dos 8 alunos, receberia posteriormente à mudança reclamações de um conjunto de alunos de tamanho quase igual, pedindo que fosse retratada ou que mudada para um terceiro novo horário. O RD teria então de voltar atrás com o professor que aceitou a mudança ou gerar uma outra mudança sem saber quantos conflitos essa nova mudança causaria. Com a solução presente, o processo ganha previsibilidade e é possível quantificar qual o valor relativo de dois horários sugeridos sem sequer pedir mudanças de horário ao corpo docente.

2.3.1 Dificuldades Encontradas

Dificuldades com a Implementação

Foram discutidas, durante a descrição dos objetos, certas dificuldades decorrentes da implementação dos objetos em C++, das quais a principal foi a manutenção de uma identidade única para todo objeto, permitindo a distinção entre todos os objetos do programa. Em essência, esse problema surge de uma necessidade conhecida como Regra de Três (do inglês *Rule of Three*) que diz que, quando se é necessário implementar um destrutor diferente do padrão para um objeto, em geral é necessário também implementar um construtor de cópia (Stroustrup, 2000). A interação entre o construtor padrão, o construtor de cópia, e o destrutor causou dúvida significativa e só foi resolvida depois de alguma pesquisa.

Em essência, a solução inicial para a geração de identidades distintas foi um contador estático (uma variável de classe `static ll count` que seria incrementada toda vez que um objeto novo fosse gerado. Dessa maneira, durante a construção do objeto, esse contador é incrementado e o valor corrente associado à id do objeto. Isso garante que, toda vez que um objeto novo for criado, sua ID será única.

Naturalmente, durante a execução do programa, podemos querer deletar um objeto. Desse modo, precisamos de um construtor não-padrão que decremente a variável estática, para manter a coêrencia entre ID e número de objetos. Isso em princípio é desnecessário, contudo por causa do jeito que C++ trata a passagem de objetos em funções, se torna necessário.

Quando um objeto é passado como valor em uma função, C++ chama o construtor de cópia do objeto para construir um desses objetos. Se um construtor de cópia não for implementado, o *default* é usar o construtor padrão passando os valores que o objeto possui. Contudo, quando o objeto é destruído ao final da função, o destrutor é chamado. Desse modo, se um construtor de cópia não for implementado, o destrutor diminuirá o Id global toda vez que uma função termina a execução, causando um valor de Id inconsistente. Dessa forma é necessário implementar um construtor de cópia especial que também incrementa o Id, garantindo que todos os valores se mantém consistentes.

Outra dificuldade significativa encontrada com a biblioteca padrão foi o comportamento do *container* `set`. Há duas particularidades: Elementos de um `set` são únicos e são imutáveis. Ambas as características foram descobertas durante a implementação, e necessitaram a definição de comparadores para instruir o `set` a ordenar e incluir consistentemente os objetos, pois o comparador padrão não funciona para objetos que não sejam primitivos, e foi necessário alterar o desenho de diversas funções de manipulação para alterar os elementos do conjunto. Essencialmente, ao invés de manipular referências aos elementos, é necessário retirar o elemento do conjunto, copiar seus valores localmente, operar sobre eles, criar um novo elemento e reinserir no conjunto, pois não é possível alterar valores dentro de um `set`.

Dificuldades como essas mencionadas, que foram as maiores, refletem principalmente inexperiência com a linguagem e foram fonte significativa de retrabalho durante a implementação do programa.

Dificuldades de *Design*

A principal dificuldade encontrada no desenvolvimento do projeto foi a quantidade de retrabalho necessária, que consumiu muito tempo e levou a cortes nas funcionalidades pretendidas. O sistema foi desenvolvido inteiramente do zero, e usa apenas a biblioteca padrão de C++ para I/O e algumas estruturas de dados. A metodologia de *design* escolhida foi do tipo *bottom-up*, em que os objetos básicos foram desenhados e testados e em seguida seriam integrados na solução final.

Contudo, na fase de integração, a limitação dessa filosofia de *design* ficou aparente, pois não era possível integrar os diversos objetos para produzir a funcionalidade desejada. Em razão disso, foi necessário retrabalho e alteração dos objetos originais para que a funcionalidade desejada pudesse ser implementada. Um ponto importante de dificuldade

foram as funções de comparação entre objetos, que foram redefinidas múltiplas vezes para implementar funcionalidades desejadas.

Consideramos que esse problema tenha sido causado principalmente por inexperiência do programador em desenvolver um sistema com múltiplas partes, mas acreditamos que uma filosofia de *design* no estilo *top-down*, em que a funcionalidade desejada é desenhada e claramente definida primeiro, levaria a menos retrabalho dos componentes e maior agilidade na produção do projeto.

Capítulo 3

Conclusão

Em conclusão, podemos dizer que a solução implementada representa uma melhora significativa nos processos de definição dos horários, especialmente para o RD, em relação aos métodos anteriores em uso, que se baseavam em metodologias *ad hoc* para decidir sobre qualquer mudança de horário, de modo que mudanças sem valor quantificado eram comuns e causavam grande quantidade de retrabalho ao RD.

Consideramos ainda que, caso uma solução automática capaz de produzir um horário fosse implementada no BCC, talvez usando o paradigma ASP que discutimos na revisão bibliográfica, nossa solução ainda teria valor como um refinador de horários, que permitiria não só ao RD avaliar o valor relativo de cada mudança e solução do ponto de vista dos alunos, mas permitiria ao produtor do horário refinar a solução também, de modo a aumentar o valor do produtor automático de horários.

A principal dificuldade encontrada no projeto foi retrabalho significativo causado por uma falta de compreensão *a priori* de quais seriam as consequências de certas decisões de desenho do projeto. Tanto a filosofia de design *bottom-up* quanto o desconhecimento de certas premissas da biblioteca padrão de C++ causaram retrabalho significativo e significaram que componentes tiveram de ser redesenhados, alterando também a função de suas dependências.

Para que a solução proposta alcance seu potencial máximo, é necessário implementar uma camada de interação com o usuário que permita interação via *mouse* e maior facilidade no cadastro e alteração de horários de disciplinas, além de ferramentas de consulta mais refinadas. Essas são as direções de desenvolvimento que pretendemos seguir para entregar uma solução mais útil à comissão de horários.

Referências

[*Administrador de Conflitos* 2023] *Administrador de Conflitos*. URL: https://github.com/EduBrancher/TCC-Administrador-de-Conflito (acesso em 23/01/2023) (citado na pg. 16).

[Aubin e Ferland 1989] Jean Aubin e Jacques A. Ferland. "A large scale timetabling problem". *Computers & Operations Research* 16.1 (1989), pp. 67–77. ISSN: 0305-0548. DOI: https://doi.org/10.1016/0305-0548(89)90053-1. URL: https://www.sciencedirect.com/science/article/pii/0305054889900531 (citado nas pgs. 8, 9).

[Banbara *et al.* 2013] Mutsunori Banbara, Takehide Soh, Naoyuki Tamura, Katsumi Inoue e Torsten Schaub. "Answer set programming as a modelinglanguage for course timetabling". *Theory and Practice of Logic Programming* 13 (2013) (citado nas pgs. 11–13).

[Boland *et al.* 2008] Natashia Boland, Barry D. Hughes, Liam T.G. Merlot e Peter J. Stuckey. "New integer linear programming approaches for course timetabling". *Computers & Operations Research* 35.7 (2008). Part Special Issue: Includes selected papers presented at the ECCO'04 European Conference on combinatorial Optimization, pp. 2209–2233. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2006.10.016. URL: https://www.sciencedirect.com/science/article/pii/S0305054806002784 (citado na pg. 10).

[Burke *et al.* 1994] E. K Burke, D. G. Elliman e R. Weare. "A university timetabling system based on graph colouring and constraint manipulation". *Journal of Research on Computing in Education* (1994) (citado nas pgs. 6, 7).

[Cooper e Kingston 1993] Tim B. Cooper e Jeffrey H. Kingston. "The solution of real instances of the timetabling problem". *The Computer Journal* 36 (1993) (citado nas pgs. 5–8).

[Cooper e Kingston 1995] Tim B. Cooper e Jeffrey H. Kingston. "The complexity of timetable construction problems". *Lecture Notes in Computer Science* 1153 (1995) (citado nas pgs. 4, 5).

[Garey e Johnson 1979] M. R. Garey e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman e Co., 1979 (citado na pg. 5).

[*Google Forms* 2023] *Google Forms*. URL: https://www.google.com/forms/about/ (acesso em 23/01/2023) (citado na pg. 15).

[Gotlieb 1962] C. Gotlieb. "The construction of class-teacher time-tables". *IFIP Congress* (1962) (citado na pg. 4).

[Karp 1972] R. M. Karp. "Reducibility among combinatorial problems". *Complexity of Computer Computations* (1972), pp. 85–103 (citado na pg. 4).

[Lifschitz 2008] Vladimir Lifschitz. "What is answer set programming?" In: AAAI'08. Chicago, Illinois: AAAI Press, 2008, pp. 1594–1597. ISBN: 9781577353683 (citado na pg. 11).

[Sammut *et al.* 2021] Roberta Sammut, Odette Griscti e Ian J Norman. "Strategies to improve response rates to web surveys: a literature review". *International Journal of Nursing Studies* (2021) (citado na pg. 26).

[Stroustrup 2000] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000 (citado na pg. 26).

[Welsh e Powell 1967] D. J. A. Welsh e M. B. Powell. "An upper bound for the chromatic number of a graph and its application to timetabling problems". *The Computer Journal* 10 (1967), pp. 85–86 (citado na pg. 4).