Noname manuscript No.
(will be inserted by the editor)

Algoritmo GRASP para o problema de tabela-horario de universidades

Walace Rocha · Maria Boeres · Maria Rangel

Received: date / Accepted: date

Abstract The timetabling problem is of great interest in the combinatorial optimization field. Given a set of disciplines, students, teachers and classrooms, the problem consists in to allocate lectures in a limited number of timeslots and rooms, respecting some restrictions. The formulations are varied, which sometimes makes it difficult to compare to other studies. Despite the differences, it is classified into three main classes: exams timetabling, schools timetabling and universities timetabling. This work specifically treats the universities timetabling and is adopted the third formulation of international timetabling competition - ITC-2007. The problem is solved with the GRASP metaheuristic. Hill Climbing and Simulated Annealing are used as local search phase of the algorithm and Path-relinking is implemented to improve the basic version. Tests were carried out simulating the same competition rules and the results are competitive with those obtained by the ITC-2007 finalists.

Keywords Educational timetabling · GRASP · Local Search

1 Introduction

Os problemas de *scheduling* tratam a alocação de recursos em determinados horários satisfazendo algumas restrições. Os problemas de tabela-horário são um subconjunto importante desta área e as aplicações são variadas como, por

W. Rocha
first address
Tel.: +55-27-40092255
E-mail: walacesrocha@yahoo.com.br

M. Boeres
second address

M. Rangel
third address

exemplo, elaboração de escala de funcionários e agendamento de partidas para campeonatos esportivos.

No caso específico de tabela-horário de escolas, o problema consiste em alocar um conjunto de aulas em um número pré-determinado de horários, satisfazendo diversas restrições envolvendo professores, alunos e o espaço físico disponível. A solução manual deste problema não é uma tarefa trivial e as instituições de ensino precisam resolvê-lo anualmente ou semestralmente. Nem sempre a alocação manual é satisfatória, visto que é difícil contemplar todos os anseios das partes envolvidas.

Por esta razão, atenção especial tem sido dada a solução automática de tabela-horário. Nos últimos sessenta anos, começando com [Gotlieb(1962)], este problema ganhou grande destaque na área de otimização combinatória, tendo diversos trabalhos publicados [Schaerf(1995),Lewis(2007)].

O problema de tabela-horário está entre os mais difíceis da área de otimização combinatória. Sua dificuldade aumenta à medida em que são adicionadas restrições. Em [Schaerf(1995)] pode ser visto que ele é classificado como NP-completo para a maioria das formulações. Assim, a solução ótima só pode ser garantida para instâncias bem pequenas, que não correspondem às instâncias reais da maioria das instituições de ensino.

Devido à complexidade do problema, métodos exaustivos são descartados. Por serem relativamente simples de implementar e produzirem bons resultados, diferentes meta-heurísticas tem sido aplicadas, com destaque para *Simulated Annealing* [Ceschia et al(2011)Ceschia, Di Gaspero, and Schaerf], Algoritmo Genético [Erben and Keppler(1995)] e Busca Tabu [Elloumi et al(2008)Elloumi, Kamoun, Ferland, and Dammak].

Este trabalho trata o problema de tabela-horário de universidades usando a meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedures*). Pelo fato de existirem diversas formulações para o problema, escolhemos a terceira formulação proposta no ITC-2007 (*International Timetabling Competition - 2007*) [PATAT(2008)]. A razão principal desta escolha é facilitar a comparação dos resultados com outros algoritmos propostos na literatura.

2 Problem Formulation

A formulação três do ITC-2007 é baseada em casos reais da Escola de Engenharia da Universidade de Udine na Itália, mas se aplica em muitas outras universidade européias. Algumas simplificações foram feitas para o campeonato para manter certo grau de generalidade. Ela possui os seguintes parâmetros:

- **Dias, Horários e Períodos**: É dado o número de dias na semana em que há aula (geralmente cinco ou seis). Um número fixo de períodos de aula, igual para todos os dias, é estabelecido. Um horário é um par composto de um dia e um período. O total de horários é obtido multiplicando a quantidade de dias pela quantidade de períodos no dia.
- **Disciplinas**: Cada disciplina possui uma quantidade de aulas semanais que devem ser alocadas em períodos diferentes. É lecionada por um professor e assistida por um dado número de alunos. Um número mínimo de dias é

determinado para a distribuição de suas aulas na semana e é possível que um professor lecione mais de uma disciplina.
- **Salas**: Cada sala possui uma capacidade diferente de assentos.
- **Currículo**: Um currículo é um grupo de disciplinas que possuem alunos em comum.
- **Indisponibilidades**: Alguns períodos são indisponíveis para determinadas disciplinas.

Uma solução consiste na alocação de cada aula em um horário e uma sala. A partir dos parâmetros são estabelecidas as restrições fortes e fracas. As fortes devem ser sempre respeitadas. Qualquer violação de uma restrição forte gera uma tabela-horário inviável, que na prática não pode ser utilizada. Por outro lado as restrições fracas devem ser satisfeitas o máximo possível, e quanto menos violações, melhor é a tabela-horário. As restrições do problema são descritas a seguir:

2.1 Hard Constraints (RFt)

- **Lectures**: Todas as aulas das disciplinas devem ser alocadas e em períodos diferentes. Uma violação ocorre se uma aula não é alocada. (RFt1)
- **Conflicts**: Aulas de disciplinas do mesmo currículo ou lecionadas pelo mesmo professor devem ser alocadas em períodos diferentes. (RFt2)
- **Room Occupation**: Duas aulas não podem ocupar uma sala no mesmo horário. (RFt3)
- **Availability**: Uma aula não pode ser alocada num horário em que a disciplina é indisponível. (RFt4)

2.2 Soft Constraints (RFc)

- **Minimum Working Days**: As aulas de cada disciplina devem ser espalhadas por uma quantidade mínima de dias. Cada dia abaixo do mínimo é contado como uma violação. (RFc1)
- **Isolated Lectures**: Aulas do mesmo currículo devem ser alocadas em períodos adjacentes. Cada aula isolada é contada como uma violação. (RFc2)
- **Room Capacity**: O número de alunos da disciplina deve ser menor ou igual ao número de assentos da sala em que a aula for alocada. Cada aluno excedente contabiliza uma violação. (RFc3)
- **Room Stability**: Todas as aulas de uma disciplina devem ser alocadas na mesma sala. Cada sala distinta é contada como uma violação. (RFc4)

Na contagem total das violações fracas são considerados pesos diferentes para cada tipo de violação. A restrição de dias mínimos possui peso 5 (cinco), aulas isoladas, peso 2 (dois) e as demais, peso 1 (um).

Uma solução viável deve atender a todas as restrições fortes. Uma solução ótima é viável e minimiza a função objetivo apresentada na equação 1:

$$f = \text{Violações}_{RFt} + \text{Violações}_{RFc} \quad (1)$$

onde $\text{Violações}_{RFt} = |RFt1|_v + |RFt2|_v + |RFt3|_v + |RFt4|_v$, $\text{Violações}_{RFc} = 5|RFc1|_v +$
$2|RFc2|_v + |RFc3|_v + |RFc4|_v$ e $|.|_v$ representa o número de violações.

3 Algoritmo GRASP para o Problema de Tabela-horario

A meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedures*) foi introduzida por [Feo and Resende(1989)] para tratar o problema de cobertura de conjuntos. Desde sua proposta inicial, o GRASP já foi aplicado com sucesso em vários problemas de otimização como conjunto independente máximo [Feo et al(1994)Feo, Resende, and Smith], problema quadrático de alocação [Li et al(1994)Li, Pardalos, and Resende], satisfabilidade [Resende and Feo(1996)], planarização de grafos [Resende and Ribeiro(1997)], roteamento de circuitos virtuais [Resende and Ribeiro(2003)], entre outros.

O algoritmo GRASP é um procedimento com iterações independentes, onde cada iteração constrói uma solução inicial e aplica busca local para melhorá-la. A resposta final é a melhor obtida dentre as iterações. O algoritmo 1 apresenta o pseudo-código genérico do GRASP. Ao final da fase de construção inicial, pode ocorrer de a solução obtida ser inviável. Por isso um passo intermediário é previsto para reparar a solução, tornando-a viável.

Algoritmo 1: Estrutura básica do algoritmo GRASP

Entrada: MaxIter
Saída: Solução S^*

```
1  f* ← ∞ ;
2  para i ← 1 até MaxIter faça
3  |  S ← GeraSolucaoInicial() ;
4  |  se S é inviável então
5  |  |  ReparaSolucao(S);
6  |  fim se
7  |  S ← BuscaLocal(S) ;
8  |  se f(S) < f* então
9  |  |  S* ← S ;
10 |  |  f* ← f(S) ;
11 |  fim se
12 fim para
```

O método de construção da solução inicial do GRASP é guloso, aleatório e visa produzir um conjunto diversificado de soluções iniciais de boa qualidade para a busca local. Algoritmos totalmente aleatórios conseguem essa diversificação, mas as soluções em geral são ruins. Por outro lado, algoritmos gulosos tendem a gerar soluções de melhor qualidade, mas eles não conseguem produzir soluções diferentes já que a construção é sempre feita por escolhas gulosas.

3.1 Geração de tabela-horário inicial

O objetivo da etapa de construção inicial é produzir uma tabela-horário viável, e se possível, com poucas violações das restrições fracas. O GRASP não exige que a solução inicial seja viável, mas foi decidido implementar desta forma para que nas fases seguintes o algoritmo concentre apenas na eliminação das violações das restrições fracas. A contagem de violações fortes e fracas requer certo esforço computacional. Garantindo que as soluções são viáveis, a etapa de busca local não necessita contar as violações fortes.

Partindo de uma tabela-horário vazia, as aulas são acrescentadas uma a uma até que todas estejam alocadas. A escolha é tanto gulosa (para produzir soluções de boa qualidade) quanto aleatória (para produzir soluções diversificadas).

Com intuito de obter uma solução viável, é adotada uma estratégia de alocar as aulas mais conflitantes primeiro. Poucos horários são viáveis para as disciplinas mais conflitantes, portanto, é melhor alocá-las quando a tabela está mais vazia. Para medir se uma aula é mais conflitante que outra são contados quantos horários disponíveis são adequados para alocar a aula da disciplina. Esta contagem envolve:

- contar a quantidade de horários disponíveis (desocupados)
- retirar os horários em o que o professor da disciplina já leciona alguma aula;
- retirar os horários em que estão alocadas disciplinas do mesmo currículo;
- retirar os horários que são indisponíveis para a disciplina segundo a restrição de indisponibilidade.

Em cada iteração, a aula mais difícil (a que possui menos horários viáveis) é escolhida para ser alocada. Existem diferentes combinações de horários e salas para a alocação. Os custos de todas essas combinações são calculados levando-se em conta as penalizações das restrições fracas. As combinações que possuem horários inviáveis são descartadas. Com base no menor e maior custo de adição de um elemento à solução (c^{min} e c^{max}) é construída a lista restrita de candidatos (LRC). Pertencem à LRC as aulas cujos custos estejam no intervalo $[c^{min}, c^{min} + \alpha(c^{max} - c^{min})]$. Uma aula é escolhida aleatoriamente da LRC e acrescentada à solução.

É possível que em alguns casos, ao escolher uma aula para alocar, não haja um horário que mantenha a viabilidade da solução. Para contornar esta situação foi implementado um procedimento denominado explosão. É uma estratégia que retira da tabela uma aula alocada anteriormente para abrir espaço para a aula que não está sendo possível alocar. A aula retirada volta para o conjunto de aulas não alocadas.

Para realizar a explosão, é preciso primeiro selecionar um horário que terá uma ou mais aulas retiradas. Foi decidido fazer esta escolha de horário aleatoriamente pois assim se evita problemas de ciclagem, que é o caso em que aulas ficam saindo e voltando para a lista de aulas não alocadas.

O algoritmo 2 ilustra o procedimento de geração de uma tabela-horário inicial. Ele recebe como parâmetro as aulas das disciplinas a serem alocadas na tabela que inicialmente é vazia (linha 1). Para facilitar a obtenção da aula mais conflitante durante as iterações é criada uma lista de aulas não alocadas (linha 2). Esta lista é ordenada de forma decrescente pela quantidade de conflitos, portanto a aula na primeira posição da lista é a mais conflitante.

A cada passo da construção da solução inicial a aula mais conflitante que ainda não foi alocada é selecionada na linha 5. Em seguida é verificado em quais horários pode ser alocada a aula sem gerar inviabilidades (linha 6). Esses horários formam o conjunto H. Se não houver horário disponível ocorre a explosão (linhas 7 a 10), portanto, H terá pelo menos um horário e será possível fazer a alocação. Os horários disponíveis são combinados com todas as salas e é feita uma avaliação do custo de alocação da aula na respectiva sala e horário (linha 11). Com base nos custos é construída a lista restrita de candidatos (linhas 12 a 14). A aula é então inserida na tabela numa posição escolhida aleatoriamente da LRC. A lista de aulas não alocadas é atualizada e ordenada novamente (linhas 17 e 18). Essa ordenação é necessária porque a última aula alocada poderá gerar conflitos com as aulas que ainda serão alocadas. O procedimento termina quando todas aulas estão inseridas na tabela-horário.

Algoritmo 2: Algoritmo construtivo para geração de tabela-horário inicial

Entrada: A = {conjunto de aulas}, R = {rooms set} α
Saída: Tabela-horário T

1 $T \leftarrow \emptyset$;
2 $ListaNaoAlocadas \leftarrow GeraListaNaoAlocadas(A)$;
3 $OrdenaAulasPorConflitos(ListaNaoAlocadas)$;
4 **enquanto** $|ListaNaoAlocadas| > 0$ **faça**
5 $a \leftarrow ListaNaoAlocadas[0]$;
6 $H \leftarrow \{h$ tal que h é viável para $a\}$;
7 **se** $H = \emptyset$ **então**
8 ExplodeSolucao(T, a) ;
9 $H \leftarrow \{h$ tal que h é viável para $a\}$;
10 **fim se**
11 Para todo $(s, h) \in S \times H, T[s, h] = \emptyset$, computar o custo de alocação $f(a, s, h)$;
12 $c^{min} \leftarrow min\{f(a, s, h) : (s, h) \in S \times H\}$;
13 $c^{max} \leftarrow max\{f(a, s, h) : (s, h) \in S \times H\}$;
14 $LRC \leftarrow \{(s, h) \in S \times H : f(a, s, h) \leq c^{min} + \alpha(c^{max} - c^{min})\}$;
15 Escolha aleatoriamente $(s', h') \in LRC$;
16 $T[s', h'] = a$;
17 $RetiraAula(ListaNaoAlocadas, a)$;
18 $OrdenaAulasPorConflitos(ListaNaoAlocadas)$;
19 **fim enqto**

As soluções produzidas pelo algoritmo 2 são sempre viáveis porque em cada iteração somente os horários que garantem a viabilidade são usados. No mel-

hor caso o algoritmo termina após n iterações, onde n é a quantidade de aulas a serem alocadas. Quando há explosões, um número maior que n iterações é executado porque aulas voltam para lista das não-alocadas. Experimentalmente foi verificado que também nestes casos o algoritmo converge e produz uma solução viável.

O parâmetro $\alpha\,(0 \leq \alpha \leq 1)$ regula se o algoritmo será mais guloso ou mais aleatório. Quando α é mais próximo de zero somente os elementos com baixo custo irão entrar na LRC. Este comportamento produz soluções de boa qualidade porém pouco diversificadas. Com α mais próximo de um, elementos com custo mais alto poderão também entrar na LRC. Isto introduz mais aleatoriedade à solução mas, em compensação, se perde em qualidade. O ideal é encontrar um valor intermediário que permita diversificação sem prejudicar muito a qualidade de solução que será passada para a fase seguinte de busca local.

3.2 Buscal local

No GRASP a diversificação é feita pela independência das iterações e pela aleatoriedade introduzida na solução inicial, enquanto a intensificação é feita pela busca local. Nesta fase a solução inicial é melhorada explorando sua vizinhança na busca de soluções melhores.

O GRASP não especifica qual a estratégia de busca local deve ser utilizada. No trabalho de [Feo et al(1994)Feo, Resende, and Smith] a busca local implementada é conhecida como *Hill Climbing*. É uma estratégia simples em que se explora a vizinhança enquanto são encontradas soluções melhores. Em [Souza et al(2004)Souza, Maculan, and Ochi] por exemplo, os autores utilizaram a Busca Tabu para compor a fase de melhoramento da solução inicial.

Neste trabalho fazemos uso de duas estratégias: a primeira, mais simples, do tipo *Hill Climbing* e a segunda, *Simulated Annealing*. Apresentamos primeiramente os movimentos realizados para explorar a vizinhança da solução:

- ***MOVE***: Uma aula é movida para uma posição desocupada na tabela-horário.
- ***SWAP***: Duas aulas trocam de posição na tabela-horário.

No *Hill Climbing* [Glover(1989)], a partir de uma solução inicial, em cada iteração um vizinho é gerado. Quando um vizinho com melhor valor de função objetivo é encontrado, ele passa a ser a solução atual. O algoritmo termina com N iterações sem melhora da função objetivo, onde N é parâmetro do algoritmo.

Particularmente para o ITC-2007, nos testes computacionais realizados, essa estratégia mostrou-se pouco eficiente, se prendendo facilmente em mínimos locais.

Uma modificação proposta foi trocar a busca em profundidade, tradicional no *Hill Climbing*, por busca em largura. No entanto, a busca em largura é

inviável para este problema, pois a quantidade de vizinhos de uma tabela-horário é muito grande. Assim, implementamos uma versão híbrida: em cada iteração são gerados k vizinhos e, caso haja melhora, o melhor deles passa a ser a solução atual. Controlando o valor k adequadamente esta versão consegue resultados superiores com uma eficiência compatível à busca em profundidade. Fazendo $k = 1$, tem-se o algoritmo *Hill Climbing* original.

Algoritmo 3: *Hill Climbing* com geração de k vizinhos por iteração

Entrada: Solução S , N, k
Saída: Melhor Solução S^*

```
1  i ← 0 ;
2  S* ← S ;
3  enquanto i < N faça
4  |  S' ← GeraVizinho(S*, k) ;
5  |  Δf ← f(S') − f(S*) ;
6  |  se Δf < 0 então
7  |  |  S* ← S' ;
8  |  |  i = 0 ;
9  |  fim se
10 |  i ← i + 1 ;
11 fim enqto
```

O algoritmo 3 mostra o algoritmo *Hill Climbing* modificado. Destaque para a função de geração de vizinho na linha 4. Além da solução atual, ela recebe como parâmetro o valor de k que é a quantidade de vizinhos que serão gerados. A função retorna o melhor deles, S'. O algoritmo inicializa na linha um o contador de iterações sem melhora. Se o melhor vizinho gerado na linha 4 é melhor que a melhor solução encontrada ($\Delta f < 0$), então este vizinho passa a ser a solução atual e o contador de iterações sem melhora é zerado.

Foi verificado que esta nova versão consegue explorar melhor o espaço de soluções, conseguindo tabelas-horário melhores com menos tempo de execução.

Mas levando-se em conta os resultados conhecidos na literatura, as respostas não estavam satisfatórias para a maioria das instâncias. Foi necessário investir numa estratégia mais rebuscada, que intensifique bem a solução inicial e seja capaz de escapar de mínimos locais. A opção adotada foi usar *Simulated Annealing* [Kirkpatrick(1984)]. Essa estratégia de busca é inspirada num processo de metalurgia que aquece um material e resfria de modo controlado visando diminuir seus defeitos.

O algoritmo *Simulated Annealing* (SA) possui três parâmetros principais: temperatura inicial, final e taxa de resfriamento. O algoritmo parte de uma temperatura inicial que vai sendo resfriada até chegar a temperatura final. Em cada temperatura, N_v vizinhos são gerados. Se o vizinho gerado é melhor que a solução atual, esta é atualizada. Se o vizinho for pior, ele é aceito com uma probabilidade igual a $P = e^{-\Delta f/T}$, onde Δf é a diferença de valor da função objetivo do vizinho e da solução atual e T é a temperatura atual. Quanto maior for Δf e menor a temperatura, menores serão as chances de aceitar o vizinho. O comportamento típico do algoritmo é aceitar grande diversificação

no início, quando a temperatura está alta. À medida que ela decresce, poucas pioras vão sendo aceitas e uma determinada região da busca é intensificada.

O algoritmo 4 apresenta o pseudo-código do *Simulated Annealing* para o ITC-2007.

Algoritmo 4: *Simulated Annealing* para fase de busca local

Entrada: Solução S , T_i, T_f, β, N_v
Saída: Solução S^*

```
1  T ← T_i ;
2  S* ← S ;
3  enquanto T > T_f faça
4  |  para i ← 1 até N_v faça
5  |  |  S' ← GeraVizinho(S*) ;
6  |  |  Δf ← f(S') − f(S*) ;
7  |  |  se Δf < 0 então
8  |  |  |  S* ← S' ;
9  |  |  fim se
10 |  |  senão
11 |  |  |  Gere um número aleatório p ∈ (0, 1] ;
12 |  |  |  se p < e^{−Δf/T} então
13 |  |  |  |  S* ← S' ;
14 |  |  |  fim se
15 |  |  fim se
16 |  fim para
17 |  T ← T * β
18 fim enqto
```

Tanto no algoritmo *Hill Climbing* quanto no *Simulated Annealing*, a geração dos vizinhos é feita na mesma proporção de utilização dos movimentos: 50% com *MOVE* e 50% com *SWAP*. Em [Ceschia et al(2011)Ceschia, Di Gaspero, and Schaerf], os mesmos movimentos são aplicados, mas para a segunda formulação do ITC-2007. Neste trabalho 60% dos vizinhos são gerados com *MOVE*, e os outros 40% são gerados com *MOVE* seguido de um *SWAP*. Foi verificado através de testes que esta estratégia não se adapta tão bem à terceira formulação do campeonato. Gerando 50% dos vizinhos com *MOVE* e os outros 50% somente com *SWAP* o algoritmo atinge soluções melhores e de forma mais rápida. Uma possível explicação para este comportamento é que aplicando dois movimentos seguidos, um movimento pode interferir na melhora obtida pelo outro. Além disso, movimentos que introduzem inviabilidades na solução são descartados. Assim a viabilidade da solução obtida na fase de construção inicial fica garantida.

3.3 Path-Relinking

A heurística *Path-Relinking* (PR) (religamento de caminhos) foi originalmente proposta por [Glover(1996)] como uma estratégia de intensificação na Busca

Tabu. A primeira proposta de uso do *Path-Relinking* no GRASP foi feita por [Laguna and Martí(1999)]. Essa hibridização tenta resolver uma deficiência do GRASP que é ausência de memorização entre as iterações.

A idéia básica do *Path-Relinking* é traçar um caminho ligando duas soluções que são chamadas de inicial e alvo. Para gerar este caminho, sucessivamente são inseridos atributos da solução alvo na solução inicial para que ela fique a cada iteração mais parecida com a solução alvo. A cada atributo inserido uma solução diferente é obtida. A melhor solução obtida no caminho é a resposta do algoritmo. Desta forma, o religamento de caminhos pode ser visto como uma estratégia que procura incorporar atributos de soluções de alta qualidade.

De forma genérica, o caminho percorrido pelo *Path-Relinking* é ilustrado na figura 1. A partir de uma solução inicial um caminho é percorrido até a solução alvo, gerando novas soluções intermediárias que podem ser melhores que a inicial e a alvo. Em cada iteração, verifica-se quais atributos estão diferentes na solução inicial e alvo. No exemplo da figura 1 são quatro diferenças, por isso quatro soluções podem ser geradas a partir da solução inicial. Opta-se por uma delas e continua a busca a partir da escolhida até se chegar na solução alvo.

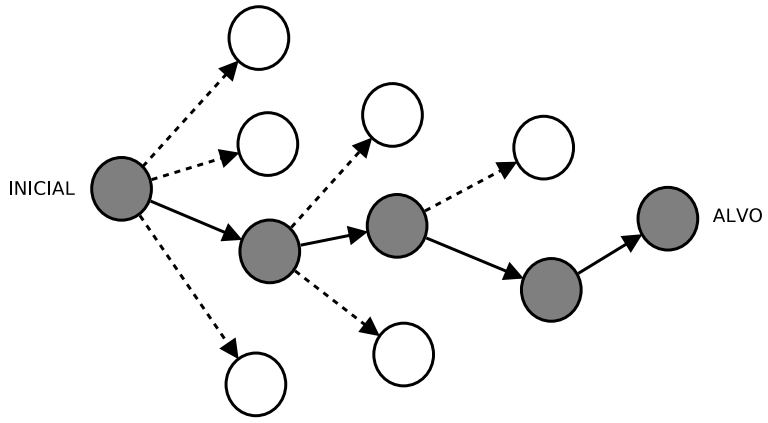

Fig. 1 Explorando soluções com *Path-Relinking*

A figura 2 ilustra o *Path-Relinking* para o caso específico de tabela-horário. Nas tabelas fictícias da figura 2 estão alocadas 7 aulas. Comparando as tabelas inicial e alvo percebe-se que 4 aulas estão na mesma posição e 3 em posições diferentes, destacadas com um círculo. No caminho percorrido, a cada nova tabela uma posição é corrigida. Todas as tabelas intermediárias são avaliadas pois podem ter melhor função objetivo.

O caminho escolhido não é único. No exemplo da figura 2, dentre as três aulas que estão em posições divergentes da solução alvo, optou-se por corrigir primeiro a aula da disciplina `AT`. Mas também poderia ter começado por outra aula. Em geral, é mais comum escolher o passo que irá produzir a mel-

hor tabela-horário (estratégia gulosa), mas pode ser introduzida aleatoriedade nesta escolha, assim como é feito na construção da solução inicial.

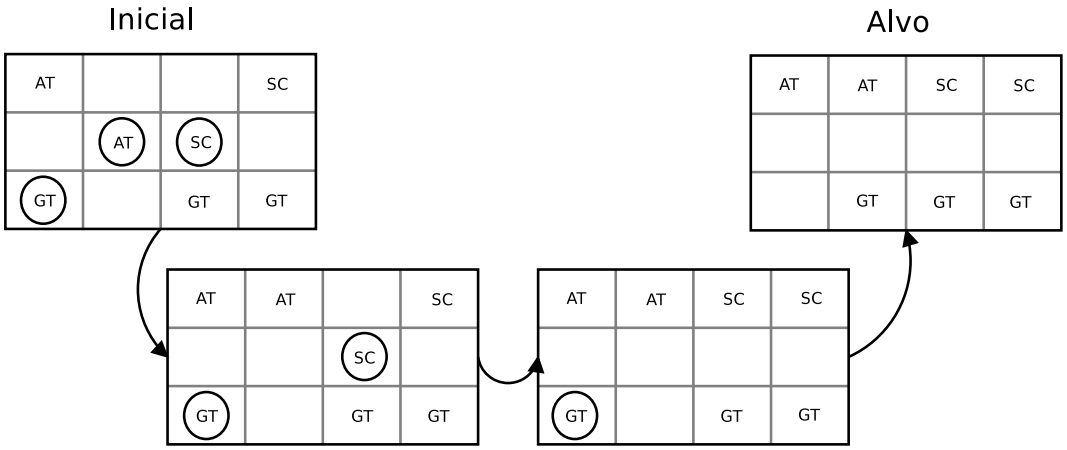

Fig. 2 Trajetória do *Path-relinking* ligando duas tabelas

O pseudo-código do algoritmo 5 ilustra o PR aplicado ao par de soluções x_i (inicial) e x_a (alvo). Nas linhas 1 e 2 são inicializadas as variáveis que guardam o melhor valor de função objetivo encontrado e a solução que produziu este valor. Na linha 4 o procedimento computa a diferença simétrica $\Delta(x_i, x_a)$ entre as duas soluções, isto é, o conjunto de movimentos necessários para alcançar x_a a partir de x_i. Um caminho de soluções é gerado ligando x_i e x_a. A melhor solução x^* no caminho é a resposta do algoritmo. Em cada iteração, o procedimento examina todos os movimentos $m \in \Delta(x, x_a)$ a partir da solução corrente x e seleciona aquele que resulta no custo de solução menor, isto é, aquele que minimiza $f(x \oplus m)$, onde $x \oplus m$ é a solução resultante da aplicação do movimento m à solução x (linhas 5 a 13). O melhor movimento m^* produz a solução $x \oplus m^*$. O conjunto de movimentos possíveis é atualizado na linha 7. Se for o caso, a melhor solução x^* é atualizada nas linhas 9 a 11. O procedimento termina quando x_a é alcançada, ou seja, $\Delta(x, x_a) = \emptyset$.

Algoritmo 5: Algoritmo *Path-Relinking*

Entrada: Solução inicial x_i, solução alvo x_a
Saída: Melhor solução x^* no caminho de x_i para x_a

$f^* \leftarrow min\{f(x_i), f(x_a)\}$;
$x^* \leftarrow argmin\{f(x_i), f(x_a)\}$;
$x \leftarrow x_i$;
Compute as diferenças simétricas $\Delta(x, x_a)$;
enquanto $\Delta(x, x_a) \neq \emptyset$ **faça**
  $m^* \leftarrow argmin\{f(x \oplus m) : m \in \Delta(x, x_a)\}$;
  $\Delta(x \oplus m^*, x_a) \leftarrow \Delta(x, x_a) - \{m^*\}$;
  $x \leftarrow x \oplus m^*$;
  se $f(x) < f^*$ **então**
    $f^* \leftarrow f(x)$;
    $x^* \leftarrow x$;
  fim se
fim enqto

O PR pode ser visto com uma estratégia de busca local mais restrita, onde os movimentos aplicados são mais específicos.

De acordo com [Resende and Ribeiro(2005)], *Path-Relinking* é uma estratégia adicionada ao algoritmo básico do GRASP, proporcionando melhoras tanto no tempo computacional quanto na qualidade de solução. Para aplicar o PR é necessário que o GRASP mantenha um conjunto de soluções elites com as melhores soluções encontradas durante a execução do algoritmo. [Resende and Ribeiro(2005)] também descrevem duas formas de inserir o PR no GRASP:

- PR é aplicado entre todos os pares de soluções elite. Pode ser feito periodicamente durante as iterações do GRASP ou no final da execução como uma pós-otimização.
- PR é aplicado como estratégia de intensificação em cada mínimo local obtido pela busca local.

No algoritmo proposto o *Path-relinking* é aplicado sobre a solução obtida na busca local e uma solução do conjunto Elite. De acordo com [Resende and Ribeiro(2005)], o caminho percorrido entre as duas soluções pode ser feito de diversas maneiras. As duas principais são:

- **Forward**: PR parte da solução pior em direção à melhor.
- **Backward**: PR é aplicado partindo da melhor solução em direção à pior.

Pode se optar por construir os dois caminhos, com a desvantagem que o tempo de execução é o dobro. [Resende and Ribeiro(2005)] indicam que no caso de fazer a opção por somente uma das trajetórias, as melhores soluções geralmente são encontradas utilizando religamento inverso. A explicação para isso é que boas soluções tendem a estar próximas à solução mais promissora. Sendo assim, o GRASP proposto neste trabalho aplica *Path-relinking* usando

uma solução elite como a inicial e a solução obtida na busca local é a solução alvo.

O algoritmo 6 resume a proposta desse trabalho: GRASP com PR para o problema de tabela-horário de universidades, terceira formulação do ITC-2007. Observe que a fase de busca local é configurável, podendo ser *Hill Climbing* ou *Simulated Annealing*, gerando duas versões diferentes.

Como consequência da introdução do Path-Relinking, um conjunto Elite se faz necessário para armazenar as melhores soluções encontradas nas iterações. Este conjunto é limitado por $MaxElite$ soluções. Ao final de cada iteração o conjunto Elite é atualizado. Todo ótimo local obtido é candidato a entrar no conjunto. Se ele já tem $MaxElite$ soluções, o candidato só irá entrar se for diferente das soluções presentes e melhor que ao menos uma delas. O controle de entrada e saída de soluções no conjunto Elite é feito pelo procedimento $AtualizaElite$.

Algoritmo 6: Algoritmo GRASP para o ITC-2007

Entrada: MaxIter, MaxElite
Saída: Melhor Solução S^*

```
1  f* ← ∞ ;
2  Elite ← ∅ ;
3  para i ← 1 até MaxIter faça
4  |  S ← GeraSolucaoInicial() ;
5  |  S ← BuscaLocal(S) // Hill Climbing or Simulated Annealing
6  |  se i ≥ 2 então
7  |  |  Selecione aleatoriamente S_elite ∈ Elite ;
8  |  |  S ← PathRelinking(S, S_elite) ;
9  |  fim se
10 |  se f(S) < f* então
11 |  |  S* ← S ;
12 |  |  f* ← f(S) ;
13 |  fim se
14 |  AtualizaElite(S, Elite) ;
15 fim para
```

4 Resultados Computacionais

As instâncias utilizadas foram as mesmas submetidas aos competidores do ITC-2007. São 21 instâncias ao todo, com grau de dificuldade variado. A organização garantiu a existência de soluções viáveis para todas as instâncias, fato que foi comprovado nos testes. Mas nada foi informado sobre a quantidade de violações fracas em cada instância. Em [PATAT(2008)] podem ser obtidas todas as instâncias.

Na tabela 1 são apresentados os dados mais relevantes de cada instância. A quantidade de horários de aula numa semana não varia muito de instância para instância. A coluna conflitos conta a quantidade de pares de aula que não

podem ser alocadas no mesmo horário (mesma disciplina, mesmo currículo ou mesmo professor) dividido pelo total de pares distintos de aula. A disponibilidade mede percentualmente a quantidade de horários que são disponíveis para as aulas, levando-se em consideração as restrições de indisponibilidade que são informadas no arquivo de entrada.

A quantidade de currículos e disciplinas tem grande impacto no tempo de execução do algoritmo, pois são mais aulas para fazer a contagem total de violações. A quantidade de conflitos e disponibilidade influencia na dificuldade de encontrar uma solução viável, pois quanto mais conflitos e menos disponibilidade, menos horários existirão para alocar aula sem violar as restrições fortes. Além de dificultar a viabilidade no momento de geração da solução inicial, os conflitos e as disponibilidades dificultam a exploração da vizinhança na busca local, dado que muitas trocas acabam sendo descartadas por introduzirem violações das restrições fortes.

Instância	**Currículos**	**Salas**	**Disciplinas**	**Horários por dia**	**Dias**	**Conflitos**	**Disponibilidade**
comp01	14	6	30	6	5	13.2	93.1
comp02	70	16	82	5	5	7.97	76.9
comp03	68	16	72	5	5	8.17	78.4
comp04	57	18	79	5	5	5.42	81.9
comp05	139	9	54	6	6	21.7	59.6
comp06	70	18	108	5	5	5.24	78.3
comp07	77	20	131	5	5	4.48	80.8
comp08	61	18	86	5	5	4.52	81.7
comp09	75	18	76	5	5	6.64	81
comp10	67	18	115	5	5	5.3	77.4
comp11	13	5	30	9	5	13.8	94.2
comp12	150	11	88	6	6	13.9	57
comp13	66	19	82	5	5	5.16	79.6
comp14	60	17	85	5	5	6.87	75
comp15	68	16	72	5	5	8.17	78.4
comp16	71	20	108	5	5	5.12	81.5
comp17	70	17	99	5	5	5.49	79.2
comp18	52	9	47	6	6	13.3	64.6
comp19	66	16	74	5	5	7.45	76.4
comp20	78	19	121	5	5	5.06	78.7
comp21	78	18	94	5	5	6.09	82.4

Table 1 Tabela com informações sobre cada instância do ITC-2007

4.1 Implementation Details

Todas as implementações foram feitas na linguagem C. A tabela-horário é representada com uma matriz, onde as linhas representam as salas e as colunas representam os horários de todos os dias. As aulas são representadas por

números inteiros. Cada aula da disciplina é representada por um número diferente. Se a primeira disciplina da instância possui cinco aulas semanais, então elas são representadas com os números 1, 2, 3, 4 e 5. Uma segunda disciplina com três aulas é representada na tabela com os números 6, 7 e 8, e assim por diante. Horários vagos são representados com o número zero.

Algumas matrizes auxiliares foram utilizadas para extrair informações de forma mais rápida. Algumas delas são estáticas pois dependem apenas das informações presentes no arquivo de entrada. Outras são dinâmicas para refletir o estado atual da tabela-horário que está sendo considerada.

Descrevemos, a seguir, as duas matrizes estáticas. Considere N o número total de aulas na instância e H o total de horários, que é a quantidade de dias multiplicada pela quantidade de períodos de aula em um dia.

A primeira matriz, chamada de AA, possui dimensão $N \times N$. Ela é utilizada para descobrir de forma rápida se duas aulas têm conflito entre si, ou se pertencem a mesma disciplina. Dadas duas aulas a_1 e a_2, adota-se a seguinte convenção:

- $AA[a_1][a_2] = 2$: as duas aulas são da mesma disciplina;
- $AA[a_1][a_2] = 1$: as duas aulas possuem conflitos entre si, seja por estarem num mesmo currículo ou por serem lecionadas pelo mesmo professor;
- $AA[a_1][a_2] = 0$: não há conflitos entre as aulas e elas não pertencem a mesma disciplina.

A segunda matriz estática, chamada de AI, possui dimensão $N \times H$. Ela é utilizada para verificar quais horários são disponíveis para as aulas segundo as restrições de indisponibilidade que são informadas no arquivo de entrada. Dois valores são possíveis nesta matriz:

- $AI[a][h] = 1$: a aula a é indisponível no horário h;
- $AI[a][h] = 0$: a aula a pode ser alocada no horário h.

É preciso ressaltar que somente $AI[a][h] = 0$ não garante a possibilidade de a ser alocada no horário h. Durante a execução do algoritmo é necessário avaliar a tabela-horário em questão para saber se existe alguma sala desocupada no horário h e/ou se nenhuma aula já alocada no horário é conflitante com a.

As matrizes dinâmicas refletem o estado da tabela-horário que está sendo considerada. O objetivo principal destas tabelas é fazer a contagem das violações fracas de forma mais eficiente. Considere C o total de currículos, DC o total de disciplinas, D a quantidade de dias, P a quantidade de períodos de aula em um dia e S o total de salas.

A primeira matriz é $DiscDias$ com dimensão $DC \times D$. Para uma dada disciplina $disc$ e um dia d, $DiscDias[disc][dia]$ retorna a quantidade de aulas da disciplina $disc$ no dia d. Com esta matriz é possível contar mais rapidamente em quantos dias há aulas de uma certa disciplina, facilitando o cálculo da contagem das violações de dias mínimos de trabalho.

A segunda matriz, $DiscSalas$, tem dimensão $DC \times S$. Para uma dada disciplina $disc$ e uma sala s, $DiscSalas[disc][s]$ retorna a quantidade de aulas

da disciplina *disc* na sala s durante a semana. Analogamente esta matriz tem por objetivo verificar quantas salas estão sendo ocupadas pela disciplina, facilitando a contagem das violações referentes à estabilidade de sala.

A terceira matriz, $CurrDiasPeriodos$, é tridimensional com tamanho $C \times D \times P$. Dados um currículo c, um dia d e um período p, $CurrDiasPeriodos[c][d][p]$ retorna a quantidade de aulas do currículo c alocadas no dia d e horário p. Esta matriz é usada para verificar se um currículo está com aulas isoladas. Se um currículo c possui alguma aula no dia d e horário p, mas $CurrDiasPeriodos[c][d][p-1] = 0$ e $CurrDiasPeriodos[c][d][p+1] = 0$ então o currículo c não tem nenhuma aula no período anterior nem no próximo, o que gera uma violação de aulas isoladas.

As matrizes dinâmicas são atualizadas quando um novo vizinho é gerado. Supondo que a aula a da disciplina *disc* foi movida da posição $(s1, d1, p1)$ para a posição $(s2, d2, p2)$ e que *disc* pertence ao conjunto de currículos C_a (pode ser mais de um currículo), as seguintes operações serão feitas:

$$\begin{aligned}
DiscDias[disc][d1] &= DiscDias[disc][d1] - 1 \\
DiscDias[disc][d2] &= DiscDias[disc][d2] + 1 \\
DiscSalas[disc][s1] &= DiscSalas[disc][s1] - 1 \\
DiscSalas[disc][s2] &= DiscSalas[disc][s2] + 1 \\
CurrDiasPeriodos[c][d1][p1] &= CurrDiasPeriodos[c][d1][p1] - 1, \forall c \in C_a \\
CurrDiasPeriodos[c][d2][p2] &= CurrDiasPeriodos[c][d2][p2] + 1, \forall c \in C_a
\end{aligned} \tag{2}$$

Além das matrizes auxiliares, outro detalhe importante é a geração e avaliação dos vizinhos. Num primeiro momento, para avaliar um vizinho gerado, o movimento era aplicado e a função objetivo avaliava toda a tabela-horário. Dado que *MOVE* e *SWAP* alteram apenas duas posições, é mais eficiente verificar o efeito das trocas localmente já que o restante da tabela permanece inalterado. Assim, as funções de geração de vizinhos retornam, além das posições de troca, um valor Δf. Se Δf é menor que zero, então significa que o vizinho terá um melhor valor de função objetivo.

4.2 Escolha dos parâmetros

O algoritmo GRASP possui dois parâmetros: número máximo de iterações *MaxIter* e o valor α que regula a forma de construção da solução inicial. Como foi implementado *Path-relinking*, um terceiro parâmetro, *MaxElite*, foi adicionado para limitar o tamanho do conjunto de soluções elite.

Quanto mais iterações, mais soluções o algoritmo pode explorar. Foi escolhido limitar a quantidade de iterações em 200. Mas como o campeonato exige que o algoritmo execute em aproximadamente 10 minutos, não é possível executar essa quantidade de iterações. O número máximo de soluções elite foi fixado em 20.

O parâmetro $\alpha \in [0, 1]$ é mais delicado. Se o seu valor for muito próximo de 0, o algoritmo de construção inicial tem comportamento mais guloso, produzindo soluções de boa qualidade porém pouco diversificadas. Se α é mais próximo de 1, as soluções são mais diversificadas mas com a desvantagem que elas tem um valor alto de função objetivo. Foram testados diversos valores no intervalo $[0.01, 0.5]$. Foi comprovado que quanto menor o valor de α, melhor a qualidade da solução, mas ainda longe do que é possível alcançar com a busca local. Foi decidido usar $\alpha = 0.15$, por produzir alguma diversificação nas soluções iniciais e manter uma qualidade razoável.

Os demais parâmetros são específicos das buscas locais. O algoritmo *Hill Climbing* possui o parâmetro N, que limita a quantidade máxima de iterações sem melhora na função objetivo. Um valor grande de N permite maior exploração dos vizinhos, mas consome maior tempo de execução. Como a idéia do GRASP é explorar soluções diferentes, N foi fixado em 10000. Empiricamente esse valor permite explorar bem a solução inicial sem prender muito tempo em um mínimo local. Um segundo parâmetro introduzido no algoritmo foi k, que fornece a quantidade de vizinhos que serão gerados por iteração. Através de testes preliminares, verificamos que com valor de $k = 10$, o algoritmo produz boas soluções e mantém um desempenho compatível com a versão original, isto é, $k = 1$ (apenas um vizinho). Valores maiores de k tornam o algoritmo lento e não há ganho justificável na qualidade das soluções.

O algoritmo *Simulated Annealing* possui mais parâmetros, tornando a tarefa de calibração mais difícil. Foram escolhidos parâmetros que permitem um certo grau de diversificação no início da busca e maior intensificação no final do processo. Como o algoritmo de solução inicial já fornece uma resposta com qualidade razoável, a temperatura inicial não precisa ser muito alta. Foi observado nos testes preliminares que fixar temperaturas altas com uma solução inicial de boa qualidade não ajuda muito o processo, pois permite aceitar soluções muito ruins e a melhora é observada apenas quando o resfriamento está bem adiantado. Foi observado também que, particularmente para as instâncias do ITC-2007, quando o algoritmo atinge uma temperatura abaixo de $0,01$ a busca estabiliza e raramente uma solução melhor é encontrada.

Com base nestas observações foram escolhidos os seguintes parâmetros: $T_i = 1,5$, $T_f = 0,005$, $\beta = 0,999$. Um valor de β um pouco maior poderia ser utilizado para fazer um resfriamento mais lento e explorar mais o espaço de busca, mas devido ao limitante de tempo estipulado pelo ITC-2007, o valor de β foi fixado em $0,999$. Em cada temperatura são gerados $N_v = 500$ vizinhos.

Como explicado na subseção 3.2, os vizinhos em todas as buscas locais são gerados somente com *MOVE* ou somente com *SWAP*. Os dois movimentos têm igual probabilidade de ocorrer.

O algoritmo *Path-relinking* implementado não possui parâmetros. A única decisão a tomar está relacionada a forma de ligar duas soluções. Testes confirmaram que as observações de [Resende and Ribeiro(2005)] se aplicam ao problema em questão, e o religamento inverso é superior ao direto. Sendo assim, o algoritmo parte de uma solução elite em direção a uma solução ótima local.

4.3 Análise dos resultados

Todos os algoritmos descritos neste trabalho foram implementados na linguagem C, compilados com GCC 4.1.2 e testados em máquina Linux com a distribuição Fedora Core 8, com processador Intel quad-core 2.4 GHz e 2 Gb de memória RAM.

Os organizadores do campeonato forneceram um programa executável para fazer um *benchmark* na máquina de testes dos competidores. O objetivo desse programa é informar um tempo de execução que seria equivalente nas máquinas do campeonato. Utilizando esse programa na máquina onde os testes foram realizados, foi estipulado um tempo de execução de 324 segundos.

Foi necessário adicionar nos algoritmos o critério de parada pelo tempo de execução, pois nos algoritmos apresentados nesse trabalho, o único critério considerado foi número máximo de iterações *MaxIter*.

Uma das melhorias implementadas no algortimo que foi citada na seção 4.1 é a geração e avaliação dos vizinhos de uma solução. Com o intuito de medir a eficiência desta modificação descrita foram criados dois programas para gerar uma tabela-horário inicial aleatória e, em seguida, gerar e avaliar 100000 vizinhos desta tabela. No primeiro programa a avaliação é feita levando em consideração toda a tabela. No segundo a avaliação é apenas local. Usando a instância *comp01* como parâmetro e executando cada programa três vezes, o primeiro executou as 100000 avaliações em um tempo computacional em torno de 44.996 segundos na média, enquanto o segundo programa executou em torno de 4.242 segundos na média. O *speedup* aproximado foi portanto de 10 vezes. Utilizando a instância *comp12*, que é uma instância maior, o primeiro programa executou em torno de 467.603 segundos na média, enquanto o segundo em 19.803 segundos. Nesta instância o *speedup* foi superior a 23 vezes. Observem que o ganho de eficiência foi bem significativo para a instância maior.

O algoritmo GRASP proposto foi executado para as 21 instâncias do ITC-2007. Para avaliar a eficiência dos tipos de busca local abordadas nesse trabalho, três versões do algoritmo foram testadas:

- **GHC1**: Busca local *Hill Climbing* com $k = 1$ (Somente um vizinho é gerado por iteração).
- **GHC2**: Busca local *Hill Climbing* com $k = 10$ (10 vizinhos são gerados por iteração).
- **GSA**: Busca local *Simulated Annealing*, com $T_i = 1,5, T_f = 0,005, \beta = 0,999, N_v = 500$.

Tanto em GHC1 quanto em GHC2 a quantidade máxima de iterações sem melhora é fixada em $N = 10000$.

A tabela 2 lista as melhores respostas encontradas para cada instância do ITC-2007. Os valores informados se referem apenas às violações das restrições fracas, dado que todas as soluções são viáveis. Assim como foi feito no ITC-2007, cada algoritmo foi executado 10 vezes com diferentes *seeds* para geração de números aleatórios. Todas as execuções foram limitadas em 324 segundos.

Instância	GHC1	GHC2	GSA	Instância	GHC1	GHC2	GSA
comp01	15	**5**	**5**	comp12	847	455	**375**
comp02	260	130	**73**	comp13	206	110	**97**
comp03	223	125	**98**	comp14	191	91	**72**
comp04	168	73	**48**	comp15	218	141	**101**
comp05	707	525	**409**	comp16	233	96	**69**
comp06	293	116	**75**	comp17	271	127	**105**
comp07	266	68	**36**	comp18	179	113	**102**
comp08	186	77	**58**	comp19	238	122	**87**
comp09	269	144	**119**	comp20	356	106	**88**
comp10	245	68	**41**	comp21	301	176	**136**
comp11	9	**0**	**0**	Média	270,52	136,57	**104,47**

Table 2 Melhores respostas obtidas pelas três versões do algoritmo: GHC1, GHC2 e GSA

Na tabela 2 pode ser observado que o algoritmo GSA é o melhor dentre as três versões descritas, sendo superior em 19 instâncias e empatando com GHC2 nas instâncias *comp01* e *comp11*. Na média, GSA supera GHC1 em 258% e GHC2 em 30%.

Nas tabelas 3 e 4 podem ser vistos os resultados obtidos pelos competidores. As colunas estão ordenadas pelas médias das respostas obtidas. Em cada tabela foi adicionada uma coluna com os resultados obtidos pelo melhor algoritmo implementado neste trabalho, o GSA. A melhor resposta de cada instância está destacada em negrito.

A competição foi dividida em duas fases descritas a seguir. A primeira fase foi mais geral e durou aproximadamente 6 meses. Todos os competidores registrados receberam sete instâncias inicialmente e mais sete faltando duas semanas para o encerramento da fase. Eles submeteram os algoritmos e as melhores respostas encontradas para cada instância. A tabela 3 lista os resultados desta primeira fase.

Os cinco melhores algoritmos da fase inicial (finalistas) foram selecionados para a segunda fase. Nesta segunda etapa os organizadores executaram os cinco melhores algoritmos com mais sete instâncias chamadas de ocultas por não serem conhecidas previamente pelos competidores. A tabela 4 lista os resultados dos finalistas com estas instâncias. Os resultados da primeira fase foram obtidos sem direitos de publicação dos nomes dos não-finalistas, por isso estão identificados apenas como 6º lugar, 7º lugar e assim por diante. Os cinco finalistas na ordem de classificação são:

1. Tomas Müller (United States)
2. Zhipeng Lu and Jin-Kao Hao (France)
3. Mitsunori Atsuta, Koji Nonobe, and Toshihide Ibaraki (Japan)
4. Martin Josef Geiger (Germany)
5. Michael Clark, Martin Henz and Bruce Love (Singapore)

Pode-se notar que para os 17 competidores iniciais, o pior resultado do algoritmo GSA foi um sexto lugar com a instância *comp05*. As melhores respostas foram obtidas para as instâncias *comp01* e *comp11*. Considerando a

média dos resultados das 14 primeiras instâncias, GSA é pior que apenas três competidores. O mesmo fato aconteceu para as últimas sete instâncias da segunda fase.

Instância	Muller	Lu	Atzuna	Clark	Geiger	6º	7º	8º	9º	10º
comp01	**5**	**5**	**5**	10	**5**	9	23	6	31	18
comp02	43	**34**	55	83	108	154	86	185	218	206
comp03	72	**70**	91	106	115	120	121	184	189	235
comp04	**35**	38	38	59	67	66	63	158	145	156
comp05	**298**	**298**	325	362	408	750	851	421	573	627
comp06	**41**	47	69	113	94	126	115	298	247	236
comp07	**14**	19	45	95	56	113	92	398	327	229
comp08	**39**	43	42	73	75	87	71	211	163	163
comp09	103	**102**	109	130	153	162	177	232	220	260
comp10	**9**	16	32	67	66	97	60	292	262	215
comp11	**0**	**0**	**0**	1	**0**	**0**	5	**0**	8	6
comp12	331	**320**	344	383	430	510	828	458	594	676
comp13	66	**65**	75	105	101	89	112	228	206	213
comp14	53	**52**	61	82	88	114	96	175	183	206
Média	**79,21**	**79,21**	92,21	119,21	126,14	171,21	192,86	231,86	240,43	246,14

Instância	11º	12º	13º	14º	15º	16º	17º	GSA
comp01	30	114	97	112	**5**	61	943	**5**
comp02	252	295	393	485	127	1976	128034	73
comp03	249	229	314	433	141	739	55403	98
comp04	226	199	283	405	72	713	25333	48
comp05	522	723	672	1096	10497	28249	79234	409
comp06	302	278	464	520	96	3831	346845	75
comp07	353	291	577	643	103	7470	396343	36
comp08	224	204	373	412	75	833	64435	58
comp09	275	273	412	494	159	776	44943	119
comp10	311	250	464	498	81	1731	365453	41
comp11	13	26	99	104	**0**	56	470	**0**
comp12	577	818	770	1276	629	1902	204365	375
comp13	257	214	408	460	112	779	56547	97
comp14	221	239	487	393	88	756	84386	72
Média	272,29	296,64	415,21	523,64	870,36	3562,29	132338,14	107,57

Table 3 Resultados da primeira fase do ITC-2007

Considerando a média dos resultados de todas as instâncias da primeira fase, GSA teve respostas 26% inferiores aos competidores Muller e Lu (primeiros colocados) e 17% superior ao quinto colocado Clark.

Na segunda fase, GSA foi inferior as respostas de Muller em 44% e superior as respostas de Clark em 29%, e em média se posiciona em quarto lugar.

Analisando separadamente por instância, podemos ver que os melhores resultados do GSA foram para as instâncias *comp01* e *comp11*, conseguindo empatar com as melhores respostas do campeonato. Por outro lado, *comp05* e *comp12* são as instâncias em que GSA obteve as respostas com valor de função objetivo mais alto. Confrontando estas informações com a tabela 1, vemos que as duas primeiras são instâncias com maior valor de disponibilidade (superior

Instância	Muller	Lu	Atsuta	Geiger	Clark	GSA
comp15	84	**71**	82	128	119	101
comp16	**34**	39	40	81	84	69
comp17	**83**	91	102	124	152	105
comp18	83	69	**68**	116	110	102
comp19	**62**	65	75	107	111	87
comp20	**27**	47	61	88	144	88
comp21	**103**	106	123	174	169	136
Média	**68**	69,71	78,71	116,86	127	98,29

Table 4 Resultados da segunda fase do ITC-2007

a 90%), enquanto que as duas últimas são as que tem menor valor, próximo a 50%. Isto significa que somente metade dos horários são disponíveis para alocar uma aula, sem considerar outros conflitos com as aula já alocadas. Além disso, *comp05* é a instância com maior percentual de conflitos entre as aulas, superior a 20%.

Essas estatísticas são relevantes porque informam o quão difícil é a exploração do espaço de soluções. Uma alta taxa de conflitos e pouca disponibilidade fazem com que os vizinhos gerados nas buscas locais sejam na maioria inviáveis, logo, acabam sendo descartados.

5 Conclusões

Esta dissertação tratou o problema de tabela-horário para universidades usando a terceira formulação do campeonato internacional de tabela-horário ITC-2007. Foi utilizado a meta-heurística GRASP, que até então só havia sido aplicada para tabela-horário de escolas. Três algoritmos foram propostos para realizar a etapa de busca local da meta-heurística: *Hill Climbing* gerando um vizinho por iteração e o mesmo algoritmo gerando k vizinhos por iteração, além de uma terceira versão com *Simulated Annealing*. O GRASP aplicado com estas buscas locais geraram respectivamente três versões: GHC1, GHC2 e GSA.

As três versões foram testadas com as mesmas 21 instâncias utilizadas no campeonato. Foi possível empatar com a melhores respostas dos competidores para as instâncias *comp01* e *comp11*. No pior caso, com a instância *comp05*, foi alcançado um sexto lugar dentre 17 competidores.

O algoritmo GRASP (GSA) implementado produziu bons resultados, obtendo resultados competitivos para o ITC-2007. Alguns pontos positivos e negativos podem ser destacados no GRASP. Entre os pontos positivos está o mecanismo de geração da solução inicial que produz soluções viáveis levando-se em consideração os custos das violações fracas. A maioria dos algoritmos na literatura focam apenas na viabilidade da solução, o que acaba produzindo soluções iniciais com alto valor de função objetivo. Outro ponto positivo é a facilidade de ajustar o comportamento do algoritmo, isto é, mais guloso ou

mais aleatório, basta regular o parâmetro α da fase de construção da solução inicial.

Um ponto negativo observado é a falta de memória entre as iterações. *O Path-relinking* melhorou um pouco este quesito, mas os ganhos não foram relevantes.

Observando estes pontos conclui-se que o GRASP prioriza mais a diversificação que a intensificação. Para introduzir mais intensificação, é preciso dar mais tempo de execução à fase de busca local. No caso específico do problema de tabela-horário a intensificação é crucial para alcançar boas respostas.

Neste trabalho foi possível comprovar que apesar das meta-heurísticas serem algoritmos genéricos adaptáveis a diversos problemas de otimização combinatória, um entendimento mais aprofundado do problema abordado é importante para obter bons resultados. No caso do problema de tabela-horário, a eficiência do GRASP foi aumentada introduzindo matrizes auxiliares e avaliações locais na geração de vizinhos.

É importante ressaltar também a relevância das vizinhanças nos problemas de otimização. A geração dos vizinhos deve ser rápida e capaz de explorar bem o espaço de soluções. No caso dos problemas de tabela-horário, além de melhoras na função objetivo, os vizinhos devem satisfazer certas restrições, o que dificulta ainda mais a exploração da vizinhança.

6 Trabalhos futuros

Foram detectadas algumas possíveis melhorias que podem ser investigadas futuramente. A primeira delas seria a implementação de vizinhanças mais específicas. *MOVE* e *SWAP* são estruturas genéricas que podem eliminar qualquer tipo de restrição forte ou fraca. As específicas fariam movimentos mais direcionados à redução de alguma violação definida, por exemplo, espalhar aulas de uma disciplina na semana para diminuir a violação de dias mínimos de trabalho. Um primeiro estudo neste sentido já foi feito, mas os movimentos eram mais complexos, prejudicando o tempo de execução. Uma forma mais eficiente deve ser investigada.

Uma segunda modificação que pode ser feita no GRASP é o aumento de memorização entre as iterações. *Path-relinking* faz isso, mas de maneira muito restrita. Uma forma que está sendo investigada é fazer com que a geração da solução inicial aproveite a estrutura da solução da iteração anterior. No caso de tabela-horário, algumas aulas seriam pré-alocadas levando em consideração a posição em que estavam na solução anterior. Neste caso seria interessante verificar as aulas que geram violações e as que não geram violações. As aulas que não estivessem gerando violações poderiam ser mantidas na mesma posição na tabela da iteração seguinte.

7 Section title

Text with citations [**?**] and [**?**].

References

1. Ceschia, S., Di Gaspero, L., Schaerf, A.: Design, engineering, and experimental analysis of a simulated annealing approach to the post-enrolment course timetabling problem. Computers & Operations Research **39**(7), 1615–1624 (2011)
2. Elloumi, A., Kamoun, H., Ferland, J., Dammak, A.: A tabu search procedure for course timetabling at a tunisian university. In: Proceedings of the 7th PATAT Conference, 2008 (2008)
3. Erben, W., Keppler, J.: A genetic algorithm solving a weekly course-timetabling problem (1995)
4. Feo, T., Resende, M.: A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters **8** (1989)
5. Feo, T., Resende, M., Smith, S.: A greedy randomized adaptive search procedure for maximum independent set. Operations Research **42**, 860–878 (1994)
6. Glover, F.: Tabu search - part i. INFORMS Journal on Computing (1989)
7. Glover, F.: Tabu search and adaptive memory programing - advances, applications and challenges. In: Interfaces in Computer Science and Operations Research, pp. 1–75. Kluwer (1996)
8. Gotlieb, C.C.: The construction of class-teacher time-tables. In: IFIP Congress, pp. 73–77 (1962)
9. Kirkpatrick, S.: Optimization by simulated annealing: Quantitative studies. Journal of Statistical Physics (1984)
10. Laguna, M., Martí, R.: Grasp and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing **11**, 44–52 (1999)
11. Lewis, R.: A survey of metaheuristic-based techniques for university timetabling problems. OR Spectrum (2007)
12. Li, Y., Pardalos, P., Resende, M.: A greedy randomized adaptive search procedure for the quadratic assignment problem. In: Quadratic assignment and related problems, *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, vol. 16. American Mathematical Society (1994)
13. PATAT: International timetabling competition. URL: http://www.cs.qub.ac.uk/itc2007 (2008)
14. Resende, M., Feo, T.: A GRASP for satisfiability. In: D. Johnson, M. Trick (eds.) The Second DIMACS Implementation Challenge, *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, vol. 26, pp. 499–520. American Mathematical Society (1996)
15. Resende, M., Ribeiro, C.: A GRASP for graph planarization. Networks **29**, 173–189 (1997)
16. Resende, M., Ribeiro, C.: A GRASP with path-relinking for private virtual circuit routing. Networks **41**(1), 104–114 (2003)
17. Resende, M.G.C., Ribeiro, C.C.: Grasp with path-relinking: Recent advances and applications (2005)
18. Schaerf, A.: A survey of automated timetabling. ARTIFICIAL INTELLIGENCE REVIEW **13**, 87–127 (1995)
19. Souza, M.J.F., Maculan, N., Ochi, L.S.: Metaheuristics. chap. A GRASP-tabu search algorithm for solving school timetabling problems, pp. 659–672. Kluwer Academic Publishers, Norwell, MA, USA (2004)