Alocação de salas usando fluxo máximo de custo mínimo em grafos bipartidos

João Batista O. Netto, Hebert Coelho da Silva

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brasil

joaonetto901@gmail.com, hebert@inf.ufg.br

Abstract. *This article presents a solution for the assignment in UFG's context. This problem can be described as follows. Given a set of requests for rooms and a set of available rooms, the goal is to optimize the occupation of rooms in a way that each room has a capacity that satisfies each attended request. With the growing graduation and post-graduation courses and therefore the growing of subjects at UFG campus, manual assignment of classes is becoming more of a challenging and consuming task. This article offers a solution to this problem using min-cost max-flow in bipartite graphs and does an analysis of current allocation done by UFG against the algorithm allocation, providing an alternative to a manual assignment.*

Resumo. *Este artigo apresenta uma solução para o problema de alocação de salas no âmbito da UFG. O problema consiste em, dado um conjunto de pedidos para uso de salas e um conjunto de salas disponíveis, otimizar a ocupação das salas garantindo que cada sala tenha capacidade suficiente para que as pessoas fiquem sentadas. Com o aumento de cursos de graduação e pós-graduação e por conseguinte de disciplinas nos campus da UFG, alocar as disciplinas em salas tem se tornando um desafio e consumindo muito tempo pois é feito de forma manual. Este artigo oferece uma solução para o problema usando fluxo máximo de custo mínimo em grafos bipartidos e analisa a alocação atual versus a alocação feita pelo algoritmo, provendo uma alternativa para a alocação manual.*

1. Introdução

A alocação de disciplinas em salas é uma tarefa importante e que ocorre rotineiramente em diversos lugares como universidades, eventos, etc. Em geral alocar salas a todos os eventos é um tarefa complexa, sendo um problema classificado como NP-completo [Even et al. 1975]. Neste sentido, encontrar uma das melhores soluções possíveis (solução ótima) para este problema pode demorar muito tempo. O objetivo deste trabalho é implementar uma solução computacional para o caso da UFG com um bom equilíbrio entre qualidade da solução e o tempo gasto, buscando colaborar com a distribuição de salas nessa instituição que atualmente é feita de forma manual.

A alocação de salas no Campus da UFG é realizado com auxílio do SIDS, cujo manual, endereço na internet e criador podem ser encontrados na referencia [Smith 2020]. A UFG cedeu os dados das alocações de salas realizadas na UFG referentes ao período de 2012.1 até 2020.1, coletados do sistema para utilização nos testes com uma instância implementada do algoritmo. Destacamos que no primeiro semestre de 2020, foram realizados 7725 pedidos para salas no Campus da UFG. A alocação atualmente é feita de forma manual por um ou vários funcionários designados pela Pró-Reitoria de Graduação da UFG. Assim, esse processo consome tempo e recursos humanos, e foi relatado que já chegou a consumir até um mês. Com o aumento de demanda por salas e o surgimento de novos cursos esse processo tende a demorar ainda mais.

Na alocação de disciplinas em salas na UFG temos dois conjuntos, o conjunto de pedidos que contém os requisitos necessários a uma determinada turma, e o conjunto de salas disponíveis com seus recursos. Nessa versão esperamos que os horários dos pedidos já estejam preestabelecidos. Desse modo, procuramos uma atribuição de pedidos para salas, de modo que a maioria dos pedidos sejam atendidos dentro dos parâmetros estabelecidos.

Modelamos as salas e os pedidos como um grafo orientado com peso nas arestas. A solução, que é uma alocação, é obtida com um algoritmo que resolve o problema de fluxo máximo com custo mínimo. Existem diversos algoritmos que resolvem tal problema, o algoritmo que utilizamos [Kogler 2018] tem, até onde sabemos, o melhor tempo de execução, que é polinomial e oferece a solução ótima para o fluxo máximo. É importante salientar que independentemente da modelagem adotada para transformar o problema da alocação de salas no problema de fluxo máximo com custo mínimo, uma solução ótima para o segundo não obterá uma solução ótima para o primeiro pois são problemas distintos.

Vários artigos já se dedicaram a estudar o problema da alocação de recursos, visto que é um problema antigo na computação, com artigos propondo soluções desde 1968 [Foxley e Lockyer 1968]. Um teste feito na UFSM [Sales et al. 2015] obteve uma melhor distribuição de espaços comparados a alocação manual anteriormente realizada. Outros autores tentaram aplicações de programação linear [Lemos et al. 2019] e algoritmos genéticos [Burke e Newall 1999].

À implementação atual do algoritmo de fluxo máximo de custo mínimo num grafo modelado com os dados obtidos do SIDS para o primeiro semestre de 2020, conseguiu alocar 7148 pedidos em menos de um minuto, sobrando 577 pedidos que não foram alocados pois não era possível alocar dentro dos parâmetros requisitados. Os 577 pedidos

não atendidos correspondem a aproximadamente 7% do total, sendo uma solução inicial que pode ajudar na solução final.

Este trabalho está organizado como segue. Na Seção 2 serão apresentadas algumas definições necessárias ao trabalho. Na Seção 3 será detalhada a modelagem do problema em um grafo bipartido. A Seção 4 abordará a implementação do algoritmo em linguagem Python3 do conteúdo apresentado anteriormente, mostrando também os dados e resultados. Por fim, apresentaremos algumas considerações e aplicações na Conclusão.

2. Definições e conceitos básicos

Esta seção contém a terminologia básica para um bom entendimento do trabalho. Usamos a notação padrão em grafos e conceitos adicionais podem ser encontrados em [Bondy e Murty 1976]. Um *grafo* G é um par ordenado (V, E), onde V é um conjunto não vazio de elementos denominados vértices, e E um subconjunto de pares de vértices denominados arestas. Um grafo pode ser *direcionado* e é comumente denominado *Dígrafo*, neste caso cada aresta é denominada de *arco* que são pares ordenados de vértices. Quando for necessário indicar o conjunto de vértices ou arestas para um grafo G específico será utilizado respectivamente a notação $V(G)$ ou $E(G)$.

Em geral, para representar um grafo no plano são utilizados círculos como vértices, linhas ligando os vértices como arestas e linhas com setas para representar arcos. Algumas vezes em problemas práticos é necessário adicionar pesos nas arestas. Para cada aresta $e \in E$ está associado um número real $w(e)$ chamado de *peso*. Na Figura 1 apresentamos do lado esquerdo um grafo G com conjunto de vértices $V(G) = \{v_1, v_2, v_3\}$ e conjunto de arestas $E(G) = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}\}$, do lado direito um grafo direcionado D com $V(D) = \{v_1, v_2, v_3\}$, $E(D) = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$ e peso nos arcos.

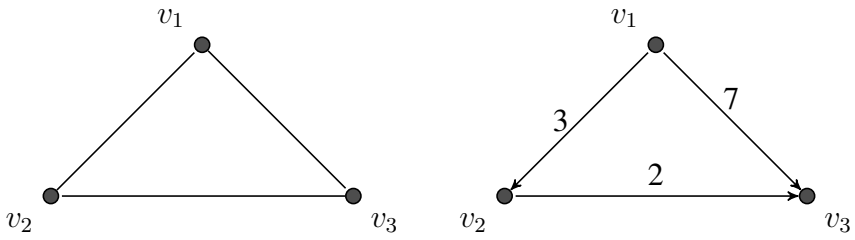

Figura 1. Grafo G e Dígrafo D com pesos nas arestas.

Um grafo é chamado de *bipartido* quando é possível particionar o conjunto de vértices em duas partições S, P de modo que não existam arestas em entre vértices de uma mesma partição. A partição (S, P) é chamada de bipartição do grafo, e S e P suas partes. Grafos bipartidos não contém ciclos ímpares, assim o grafo G da Figura 1 não é bipartido.

Um *emparelhamento* M um grafo G é um subconjunto de arestas onde não existem duas arestas incidindo no mesmo vértice. Por exemplo, sejam $x, y, z \in V(G)$, se $\{x, y\} \in M$ então não existe $\{x, z\} \in M$ ou $\{y, z\} \in M$. Desde que estamos falando de um grafo não direcionado a ordem em que os vértices aparecem nas arestas não é importante. Um emparelhamento máximo significa que não existe um subconjunto M' maior

que o conjunto M. Na Figura 2 apresentamos um grafo G com emparelhamento de arestas representados pelas linhas pontilhadas, note que este não é o único emparelhamento, mas é um emparelhamento máximo.

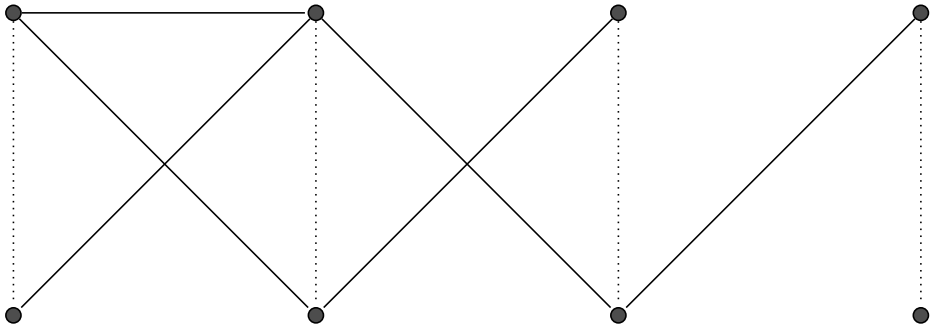

Figura 2. Exemplo de emparelhamento.

2.1. Problema de fluxo máximo com custo mínimo

Este tópico é fundamental para este trabalho, pois a solução que propomos utiliza a transformação do problema de alocação de salas em um caso de fluxo máximo em redes.

Uma *rede* é um Dígrafo $D = (V, E)$ em que cada aresta $e \in E(D)$ está associado um número real positivo $c(e)$ denominado *capacidade* da aresta e. Suponha que D possua dois vértices especiais e distintos $g, t \in V(D)$ chamados respectivamente de *origem* e *destino*, com as seguintes propriedades: o primeiro é uma fonte que alcança todos os vértices. Enquanto o destino é um sumidouro alcançado por todos. Um *fluxo* f de g a t em D é uma função que associa a cada aresta $e \in E(D)$ um número real não negativo $f(e)$ satisfazendo às seguintes condições:

- $0 \leq f(e) \leq c(e)$, para toda aresta $e \in E(D)$.
- $\sum_{w_1} f(w_1, v) = \sum_{w_2} f(v, w_2)$, para todo vértice $v \neq g, t$.

A primeira condição acima simplesmente indica que o fluxo em cada aresta não ultrapassa o valor de sua capacidade. A segunda significa que o somatório dos fluxos das arestas que entram em v é igual ao fluxo das arestas que saem de v. Este somatório é denominado valor do fluxo em v.

O valor do fluxo na origem é denominado valor do fluxo na rede D e denotado por $f(D)$. Dada uma rede D o *problema do fluxo máximo* consiste em determinar $f(D)$ máximo, ou seja, $f(D) \geq f'(D)$ para todo fluxo $f'(D)$ possível. Consideramos também além da capacidade c de cada aresta o *custo* w por unidade de fluxo em cada aresta. Assim, o objetivo do fluxo máximo com custo mínimo é encontrar um fluxo $f(D)$ máximo no qual a soma dos custos seja a mínima possível.

3. À modelagem para o problema de alocação usando redes

Conforme já citado na Introdução, no problema da alocação de disciplinas em salas existem dois conjuntos, o conjunto das salas S e o conjunto de pedidos P. Na modelagem consideramos que S e P são subconjuntos dos vértices $V(D)$ formando as duas partições

de um dígrafo bipartido D. Por simplicidade, consideramos que a sala está vazia no horário determinado pelo pedido.

O conjunto dos arcos $E(D)$ dependem exclusivamente de características específicas ao problema na UFG levantadas junto aos gestores do SIDS. Tais características estão relacionados a dados apresentados nas Tabelas 1 e 2. Assim, existe um arco $e = (p, s) \in E(D)$ entre $p \in P$ e $s \in S$ sempre que a sala referente a s for do mesmo tipo e prédio do pedido p, vejam as colunas 2 e 3 nas Tabelas 1 e 2.

Tabela 1. Conjunto de salas exemplo

Nome	Prédio	Tipo	Capacidade	Nome no Grafo
Sala 1	CA1	Sala	40	S1
Sala 2	CA1	Laboratório	20	S2
Sala 3	CA2	Sala	40	S3

Tabela 2. Conjunto de pedidos exemplo

Nome	Prédio	Tipo	Capacidade	Nome no Grafo
Pedido 1	CA1	Sala	50	P1
Pedido 2	CA1	Laboratório	15	P2
Pedido 3	CA2	Sala	40	P3

Outro dado importante é a capacidade da sala e do pedido que denotaremos respectivamente por CS e CP, tais dados estão presentes na quarta coluna das Tabelas 1 e 2. Os arcos definidos no dígrafo D recebem pesos, que são como penalidades, conforme a Equação 1. Diferentes equações podem levar a resultados distintos de alocação. A seguinte fórmula foi utilizada pelo algoritmo durante os testes:

$$Custo = |CS - CP| - 100 \cdot PR \tag{1}$$

A variável $PR \in \mathbb{R}$ na Equação 1 é a prioridade que pode ser definida junto com cada pedido para indicar que determinados pedidos devem ser atendidos com mais ou menos privilégio. Veja que, para qualquer valor de $PR > 0$, o peso do arco que é o custo de cada unidade de fluxo definido na Equação será diminuído, tornando o pedido mais atraente de ser atendido. Como a prioridade não é atualmente considerada nos dados atuais para a alocação de salas na UFG, consideramos neste trabalho nula a prioridade PR e a fórmula se reduz a $Custo = |CS - CP|$.

Na Figura 3 apresentamos o dígrafo bipartido D' inicial conforme a modelagem apresentada e obtido dos dados apresentados nas Tabelas 1 e 2. Para montar uma rede D a partir de D' conforme visto na Seção 2.1, resta adicionar dois vértices g e t, e arcos (g, p) e (s, t) para todo $p \in P$, $s \in S$. Consideramos que a capacidade de todos os arcos será $c = 1$ e para os arcos que partem de g ou chegam em t terão $Custo = 0$.

Obtemos agora uma rede de fluxo onde poderemos utilizar o algoritmo de fluxo máximo custo mínimo para obter um emparelhamento máximo. Na Figura 4 é apresentado um exemplo de rede D à esquerda, e à direita um fluxo máximo para a rede D obtido pela implementação do algoritmo utilizado neste trabalho. O fluxo passando por cada

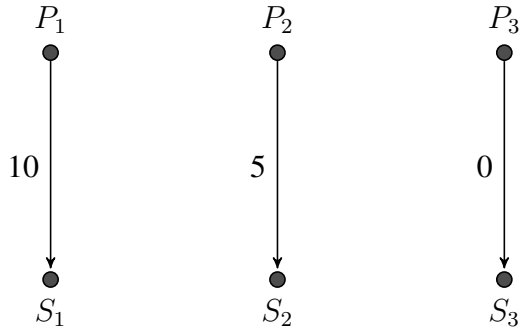

Figura 3. Dígrafo D' inicial gerado com os dados das Tabelas 1 e 2.

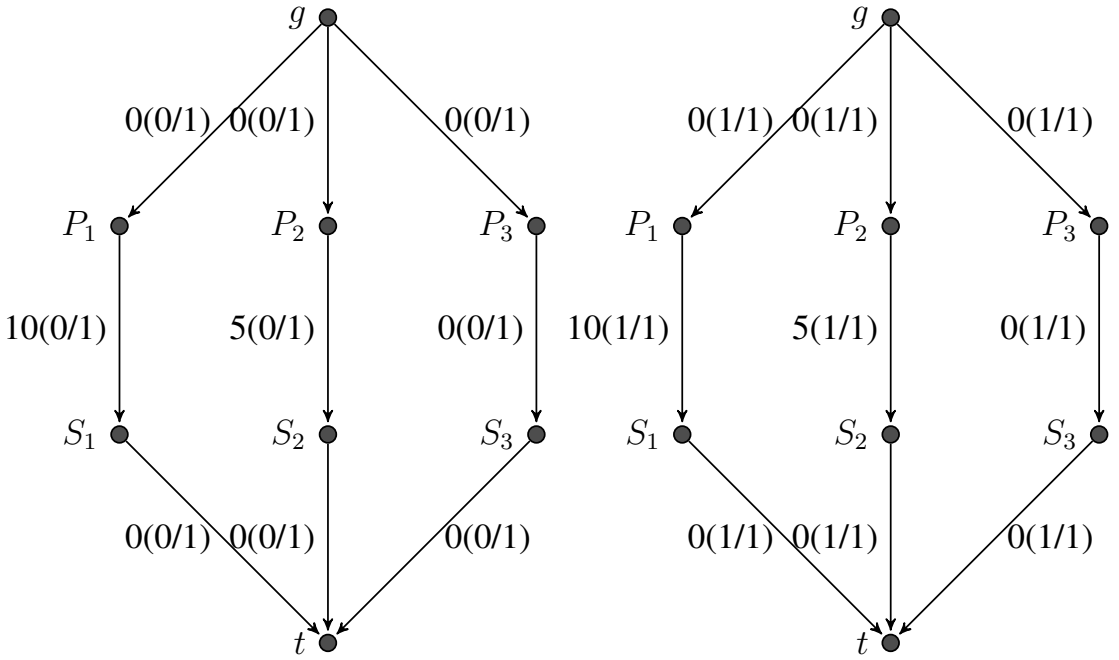

Figura 4. À esquerda uma rede D gerado a partir de D' da Figura 3. À direita um fluxo máximo para Rede D.

aresta está representado no primeiro valor dentro do parenteses. O fluxo que passa pelas arestas da solução sempre correspondem a um emparelhamento entre P e S.

Um dado a considerar é que os pedidos da UFG são divididos por horários, no total sendo 7 dias e 17 horários por dia. Para reduzir o uso de memória RAM pelo computador usado nos testes, optamos por montar $7 \cdot 17 = 119$ redes conforme ao modelado na Figura 4 e rodamos o algoritmo de fluxo máximo com custo mínimo 119 vezes. Na solução apresentada na Figura 4, a alocação foi realizada com o custo total de 15.

O objetivo da implementação que apresentamos na Seção 4 é alocar o maior número de pedidos com o menor custo. O algoritmo utilizado garante que fluxo máximo com custo mínimo é obtido para uma rede qualquer, assim a solução depende da modelagem utilizada.

4. Uma implementação do fluxo máximo com custo mínimo

A implementação aqui apresentada pode ser encontrada em [Netto 2020] e foi realizada usando a linguagem de programação $Python3$ [van Rossum et al. 2001]. Foi decidido que essa seria a melhor linguagem para integração com o SIDS e para futura manutenção do algoritmo. Dividimos a implementação em duas partes principais: a atribuição das salas e o fluxo máximo com custo mínimo.

A implementação de fluxo máximo com custo mínimo foi uma adaptação da encontrada em [Kogler 2018] e reescrita em $Python3$ e não será detalhada neste artigo por restrições de espaço. A implementação da atribuição é uma adaptação da modelagem descrita na Seção 3 e terá sua própria subseção para ser devidamente explicada, após apresentação dos dados gerados a partir do SIDS.

4.1. Dados gerados a partir do SIDS

Utilizamos os dados de salas e pedidos encontrados no SIDS para os anos de 2012 a 2020. Para cada semestre foram fornecidos dois arquivos no formato texto com a extensão txt. O primeiro arquivo é denominado "Rooms.txt"e sua primeira linha é apresentada na Tabela 3. O segundo arquivo é denominado "Lessons.txt"e sua primeira linha é apresentada na Tabela 4.

No arquivo "Rooms.txt"os campos utilizados para à modelagem tem respectivamente os mesmos nomes que os utilizados na Tabela 1: Prédio, Tipo e Capacidade. No arquivo "Lessons.txt"os campos utilizados para à modelagem tem os nomes Bld, Type e Vacan respectivamente para Prédio, Tipo e Capacidade na Tabela 2.

Nos dados obtidos, os pedidos já tem um horário específico para ser atendido, não havendo mudanças de horários de pedidos pelo algoritmo. As salas foram esvaziadas e são tratadas como se estivessem o dia inteiro livre.

Tabela 3. Arquivo Rooms.txt

Id	Prédio	Capacidade	Tipo	Special	Nome
1	1	45	1	1	101
...	...	...	...	...	...

Tabela 4. Arquivo Lessons.txt

ID	Group	Solicit	Course	Entity	Day	Hour	Bld	Type	Room	Vacanc	Matric	Priori	Special
1	1	1	81	33	3	8	4	1	125	60	51	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...

4.2. Atribuição

O SIDS divide a semana em 7 dias e 17 horários. Foi montado um grafo para cada horário e executado o algoritmo de fluxo para cada um deles, obtendo um total de 119 grafos. A função $assignPerTime$ é responsável por fazer essa separação.

Essa função recebe um conjunto de salas e pedidos de acordo com os dados da subseção anterior e cria um subconjunto P dos pedidos em que $day = lesson.day$ e $hour = lesson.hour$. Após isso ela manda esse subconjunto para a função $assignLessonsToRoom$ que irá modelar os dois conjuntos para uma rede de fluxo.

```
1  def assignPerTime(rooms, allLessons):
2      for day in range(1,8):
3          for hour in range(1, 18):
4              currentLessons = []
5                  for lesson in allLessons:
6                  if(lesson.day == day and
7                      lesson.hour == hour):
8                      currentLessons.append(lesson)
9              assignLessonsToRoom(rooms, currentLessons)
```

```
1  def assignLessonsToRoom(Rooms, Lessons):
2      edges = []
3      totalSize = len(Lessons) + len(Rooms) + 2
4      destiny = totalSize - 1
5      source = 0

7      for room in Rooms:
8          createEdge(edges, room, destiny, 0, 1)
9          for lesson in Lessons:
10             if(room.bld == lesson.bld and
11                 room.roomType == lesson.roomType):
12                 custo = abs(room.cap - lesson.vacan)
13                     - 100 * lesson.priori
14                 createEdge(edges, lesson, room, cost, 1)
15     for lesson in Lessons:
16         createEdge(edges, source, lesson, 0, 1)

18     graph = Graph(totalSize)
19     flow = graph.minCostFlow(source, destiny, edges)

21     assignment = graph.getFlowResult()
```

Essa função pode ser subdivida em três partes, tendo a inicialização entre as linhas 2 e 5. Para inicialização o número de vértices do grafo é acrescido de mais duas unidades, para acomodar os dois vértices extras que são a origem e o destino. A segunda parte, entre as linhas 7 e 16, cria arestas entre os pedidos e salas com o valor custo que foi definido anteriormente. Após as arestas do meio serem criadas, adicionamos as demais arestas entre g e o conjunto P. Finalizamos criando um grafo e aplicando o fluxo. Após isso, precisamos apenas encontrar as arestas entre pedidos e salas que tem fluxo passando por elas, obtendo assim uma alocação máxima de pedidos.

4.3. Algoritmo e Complexidade

O algoritmo de fluxo máximo com custo mínimo utilizado neste trabalho é apresentado pela função $minCostFlow$. Sua complexidade é de $\mathcal{O}(n^2m^2)$, onde n é o número de vértices e m o número de arestas. Na função $assingLessonsToRoom$ iteramos pelas S salas e pelos K pedidos para aquele horário. Como criar arestas e comparação é $\mathcal{O}(1)$, obtemos uma complexidade de $\mathcal{O}(S \cdot K)$.

Na função $assignPerTime$, iteramos 119 vezes pelos P pedidos totais, obtendo uma complexidade de $\mathcal{O}(119 \cdot P) = \mathcal{O}(P)$. Como rodamos o fluxo para cada horário, obtemos a seguinte complexidade $\mathcal{O}(119 \cdot (P + S \cdot K + n^2 \cdot m^2))$. Como 119 é uma constante, podemos removê-la, obtendo a complexidade final de $\mathcal{O}(P + S \cdot K + n^2 \cdot m^2)$.

```
1  def minCostFlow(source, destiny, edges):
2      costMatrix = [0,0]
3      capacityMatrix = [0,0]
4      createGraph(edges)
5      flow = 0
6      distance = []
7      path = []
8      while(flow < INFINITY):
9          BellmanFord(source, distance, path)
10         if(distance[destiny] == INFINITY):
11             break
12         currentFlow = INFINITY - flow
13         currentVertex = destiny
14         while(currentVertex != source):
15             j = currentVertex
16             i = path[currentVertex]
17             currentFlow = min(currentFlow, capacityMatrix[i][j])
18             currentVertex = i

20         flow += currentFlow
21         currentVertex = destiny
22         while(currentVertex != source):
23             j = currentVertex
24             i = path[currentVertex]
25             capacityMatrix[i][j] -= currentFlow
26             capacityMatrix[j][i] += currentFlow
27             currentVertex = i
28     return flow
```

O algoritmo $minCostFlow$ acima computa o maior fluxo com custo mínimo para o conjunto de arestas dada para a função. Nas linhas 2 a 7 um grafo é inicializando, junto com uma matriz de custo e outra de capacidade de tamanho $n \cdot n$ sendo n o número de vértices do grafo. Essas matrizes guardam o custo de um transicionar de um vértice i para j e a capacidade de um vértice i para j, nos índices ij, respectivamente.

É criada a rede de fluxo na linha 4 e o fluxo é inicializado com 0 na linha 5. Dois vetores, distancia e caminho, são inicializados nas linhas 6 e 7 respectivamente. Esses dois vetores correspondem, na posição i, a distancia do vértice i até o vértice "$source$" no vetor "$distance$", que é o vértice anterior ao vértice i no vetor "$path$".

Nas próximas linhas temos o algoritmo em si. É usado o algoritmo de Belmman-Ford, linha 10, para encontrar o menor caminho entre dois vértices num grafo. Nesse caso entre o vértice "$source$" e "$destiny$". Note que esse Bellman-Ford é ligeiramente modificado e considera que se a capacidade entre ij for 0, não há caminho entre os dois.

No caso de não existir um caminho entre os dois vértices, "$source$" e "$destiny$", já temos o fluxo máximo e o algoritmo encerra, linhas 11 e 12. No caso de o caminho existir, o algoritmo procede para aumentar o fluxo o máximo possível, que no caso é a menor capacidade no caminho entre a origem e o destino encontrados no algoritmo de Bellman-Ford, linhas 13 a 20. Após esse mínimo ser encontrado, ele é adicionado ao fluxo. Após isso resta finalizar diminuir a capacidade de ij e adicionar na capacidade de ji para manter a rede residual atualizada.

5. Resultados

Analisamos dados de vários semestres que podem ser observados na Figura 5. Por motivos de comparação, usamos a alocação original, em que foi rodado um programa auxiliar para verificar quantos pedidos não foram alocados. Observe que em semestres onde a alocação do algoritmo foi pior que a manual, foram apenas semestres onde não havia mais salas para o algoritmo alocar pedidos. Além disso, a alocação manual foi melhor pois ocorreu um erro na alocação manual que atribui salas aos pedidos com tipos diferentes aos requisitados. A solução obtida pela implementação sempre alocou todas as salas disponíveis e para os pedidos não atendidos não havia mais salas.

Testamos outras configurações de restrições, não criando arestas entre pedidos e salas se a capacidade do pedido excedesse ou fosse menor que 20, 50 e 75% da sala, como visto na Figura 6. Como o percentual de pedidos alocados diminui quando se aplica essas restrições, o algoritmo não usou as restrições de capacidade na hora da alocação. Todavia, os resultados com essas restrições são mostrados na Figura 6.

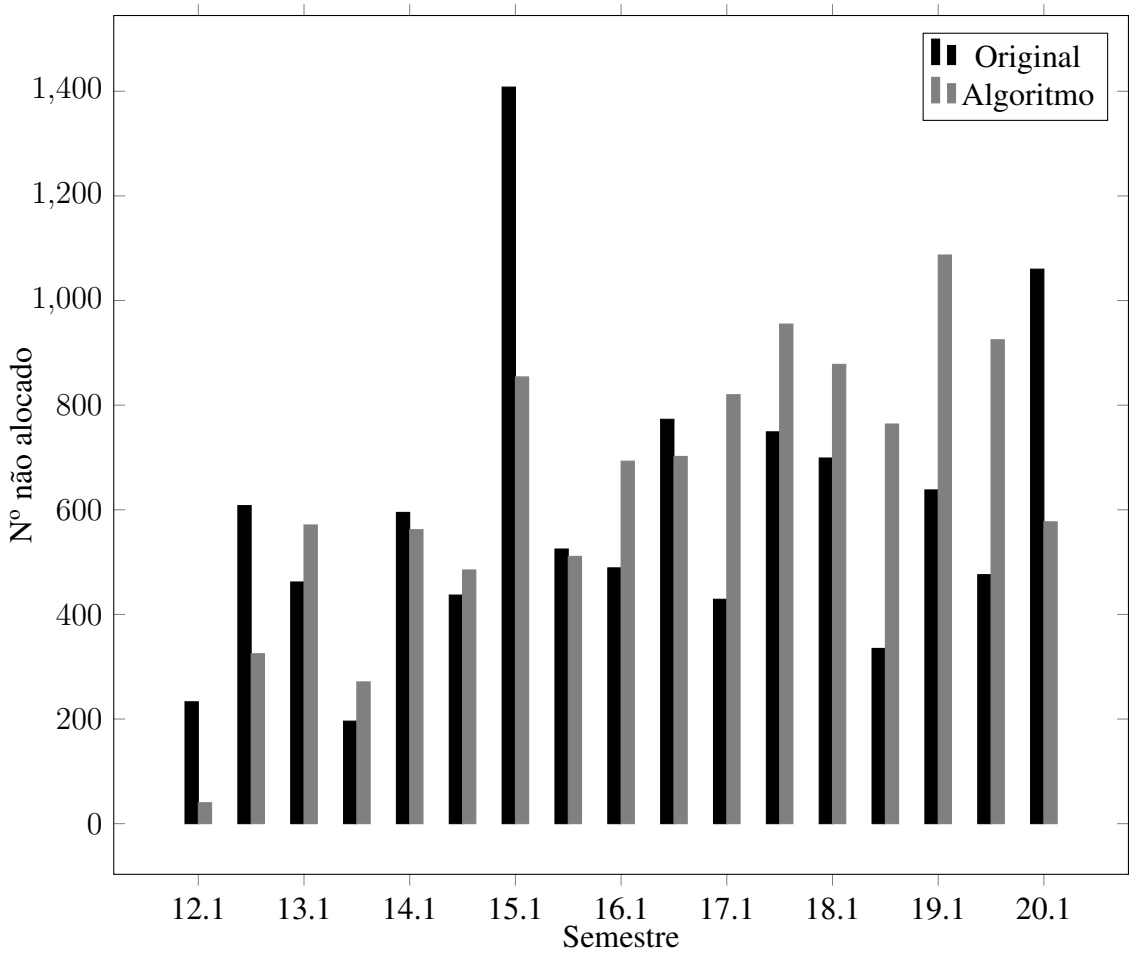

Figura 5. Pedidos alocados

Se observa que o algoritmo alocou todos os pedidos possíveis, enquanto a original teve pedidos não alocados devido a outros motivos que não temos acesso. Um pré-processamento dos dados pode levar a um resultado melhor, visto que os dados utilizados traziam pedidos para aulas de mestrado e estágio, que nem sempre recebem uma sala.

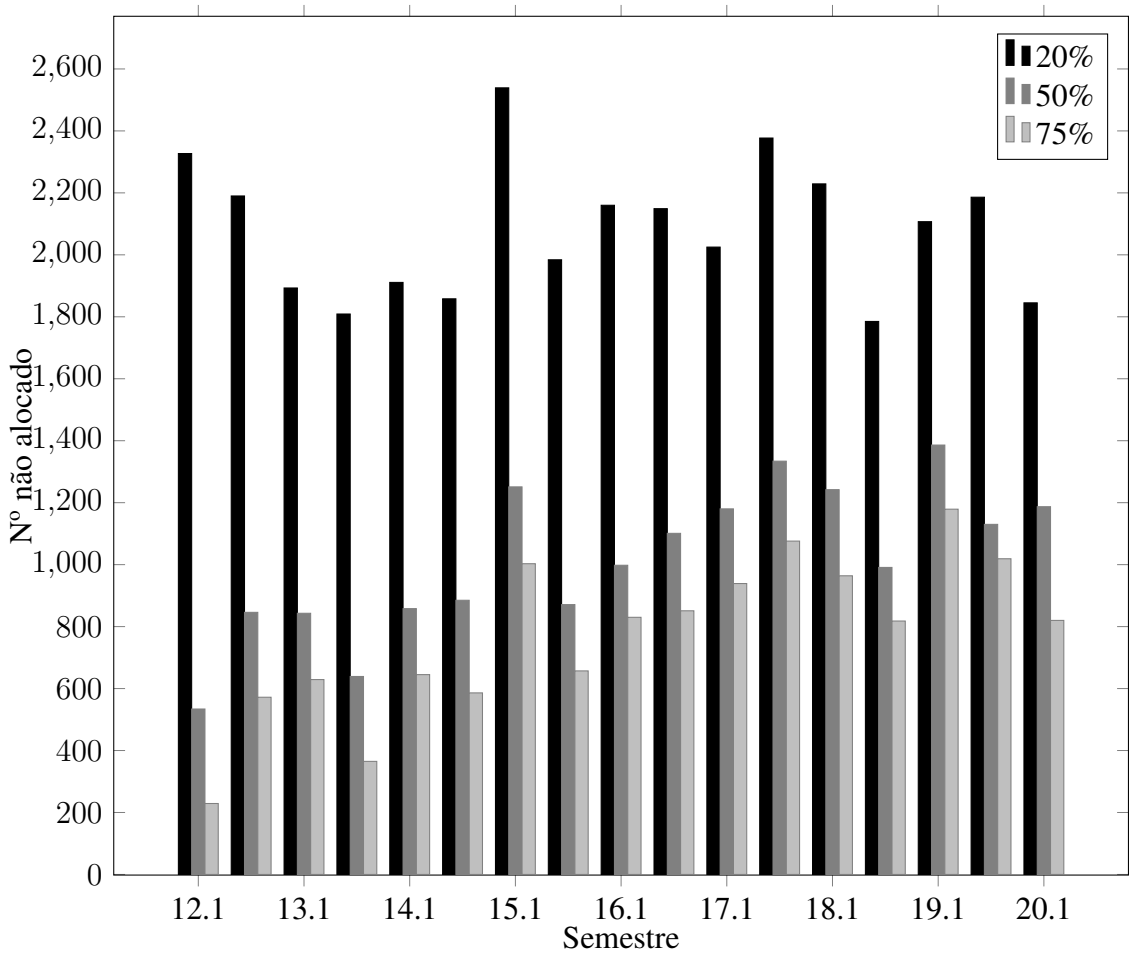

Figura 6. Pedidos alocados com diferentes percentagens.

A implementação foi executada em uma máquina com as seguintes especificações:

Ubuntu 18.04 64 bits, 4GB RAM, Intel® Core™ i5-7200U CPU @ 2.50GHz 4,Intel® HD Graphics 620 (Kaby Lake GT2).

A aplicação gerou uma solução para cada semestre com tempo não superior a 55 segundos. Vale também destacar que com os dados obtidos as soluções produzidas são alocações maximais em todos os casos, ou seja, os pedidos não atendidos são por falta de salas. Algumas considerações que são usadas na UFG não foram utilizadas neste modelo e serão desenvolvidas posteriormente, uma delas é que a alocação de alguns pedidos podem ser agrupados na mesma sala.

6. Conclusão

Essa aplicação foi desenvolvida para ser utilizada com o SIDS, podendo ser executada diretamente com os dados oferecidos no SIDS. Como a implementação do algoritmo tem o tempo de execução muito pequeno ela pode ser usada para pré-processamento, descobrindo se a alocação será válida antes do período de matrícula. Desse modo, ficaria mais fácil mudar algum pedido de horário caso não tenha sala disponível.

A implementação também pode ser utilizada para reduzir o trabalho dos servidores, que poderão se atentar apenas a resolver as disciplinas que não tiveram salas alocadas.

Visto que em todas as alocações realizadas pela implementação com os dados obtidos, no máximo 13 pedidos não foram alocados. Assim, os servidores terão menos trabalho e poderão se dedicar a outras atividades.

Outro ponto foi observado é que o programa retorna um arquivo compatível com o SIDS, que pode ser utilizado na visualização da alocação. Tendo em vista os pontos anteriores, a automatização do processo de alocação pode ser benéfica para a UFG, reduzindo o uso de mão de obra numa atividade repetitiva. Visto que o tempo de alocação do algoritmo é extremamente menor que o tempo original, podem ser testadas várias alocações diferentes, mudando o peso das arestas.

Pré-alocações manuais também podem ser feitas antes do algoritmo, para caso alguma aula precise de uma sala específica. Essas alocações podem ser feitas rapidamente, mesmo adicionando novos pedidos, a alocação antiga se manteria, tornando o processo simples. Com esses resultados e trabalhando com os mantedores do sistema, produzimos uma solução que atende as crescentes demandas da UFG gerando uma solução rápida e boa.

Referências

Bondy, J. A. e Murty, U. S. R. (1976). *Graph Theory with applications*. Elsevier Science Publishing, 1st edition.

Burke, E. e Newall, J. (1999). A multistage evolutionary algorithm for the timetable problem. *IEEE Transactions on Evolutionary Computation*, 3(1):63–74.

Even, S., Itai, A., e Shamir, A. (1975). On the complexity of time table and multi-commodity flow problems. Em *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, páginas 184–193.

Foxley, E. e Lockyer, K. (1968). The construction of examination timetables by computer. *The Computer Journal*, 11(3):264–268.

Kogler, J. (2018). Minimum-cost flow - successive shortest path algorithm. `https://cp-algorithms.com/graph/min_cost_flow.html`, Acesso em Julho de 2020.

Lemos, A., Melo, F. S., Monteiro, P. T., e Lynce, I. (2019). Room usage optimization in timetabling: A case study at Universidade de Lisboa. *Operations Research Perspectives*, 6:100092.

Netto, J. (2020). TCC em ciência da computação no INF/UFG: Implementação do problema de alocação de salas na UFG em Python. `https://github.com/joaobnetto/ICPython`, Acesso em Agosto 2020.

Sales, E. S., Müller, F. M., e Simonetto, E. O. (2015). Solução do problema de alocação de salas utilizando um modelo matemático multi-índice. Porto de Galinhas, Pernambuco - PE. Simpósio Brasileiro de Pesquisa Operacional.

Smith, O. P. (2020). Sistema de distribuição de Salas. `https://files.cercomp.ufg.br/weby/up/90/o/Tutorial_SiDS_v3.14.pdf`, Acesso em Julho 2020.

van Rossum, G., Warsaw, B., e Coghlan, N. (2001). Style guide for python code. `https://www.python.org/dev/peps/pep-0008/`, Acesso em Agosto 2020.