LIII Simpósio Brasileiro de Pesquisa Operacional
João Pessoa - PB, 3 a 5 de novembro de 2021

Redes Neurais Competitivas para o Problema Online do Escalonamento

Tiago Rafael Amaral Reis
Universidade Federal de Itajubá
tiagoamral23@gmail.com

João Victor Mendes Freire
Universidade Federal de São Carlos
joaovictormfreire@me.com

Mário César San Felice
Universidade Federal de São Carlos
felice@ufscar.br

Pedro Henrique Del Bianco Hokama
Universidade Federal de Itajubá
hokama@unifei.edu.br

RESUMO

Neste trabalho tratamos do problema do escalonamento online, no qual uma sequência de tarefas deve ser escalonada em um conjunto de máquinas idênticas, com as restrições de que as tarefas chegam uma por vez e devem ser atribuídas a alguma máquina sem o conhecimento daquelas que ainda vão chegar. Propusemos um algoritmo baseado em redes neurais para este problema. Para treinar a rede utilizada em nosso algoritmo usamos a abordagem das Redes Neurais Competitivas, implementada por meio de um algoritmo genético. Os resultados obtidos demonstram que as redes neurais, mesmo as mais simples com 2 camadas escondidas e 2 neurônios por camada, foram capazes de obter melhores resultados que o algoritmo clássico *List Scheduling*, ao menos quando as instâncias apresentam alguma heterogeneidade. Nesse caso a melhor rede obtida pela nosso método obteve soluções melhores em 81% das instâncias de teste.

PALAVRAS CHAVE. Aprendizado de Máquina, Escalonamento, Problemas Online.

Tópicos: Otimização Combinatória, Inteligência Computacional

1. Introdução

Problemas de Escalonamento possuem grande relevância prática na computação. Em Sistemas Operacionais, por exemplo, o escalonamento de processos e tarefas permite que diferentes aplicações compartilhem os recursos de um computador de maneira eficiente, sendo a ferramenta que permite a execução de várias aplicações de modo aparentemente simultâneo em um sistema com um único processador. A relevância destes problemas se estende a outros domínios, como a Pesquisa Operacional. Nesse contexto, o escalonamento é uma ferramenta importante para diminuir os prazos e custos de fabricação de manufaturados, decidindo como distribuir as tarefas entre as várias linhas de produção de uma indústria, que são compatíveis com essas tarefas.

A primeira aparição de um algoritmo com tempo polinomial e garantia de qualidade da solução em razão do ótimo foi em 1966, quando Graham [1966] descreveu o algoritmo guloso *list scheduling*, uma 2-aproximação para o problema do escalonamento em m máquinas idênticas

com o objetivo de minimizar o *makespan*. Finn e Horowitz [1979], em 1979, apresentaram um algoritmo de busca local para este problema, e mostraram também se tratar de uma 2-aproximação. Em 1969, Graham [1969] propôs uma 4/3-aproximação com uma variação do *list scheduling*, que usa a regra *longest processing time*. Ele ainda apresentou uma família de algoritmos que são um PTSA (Polynomial-Time Approximation Scheme) para minimizar o *makespan* quando o número de máquinas é constante. Hochbaum e Shmoys [1987] exibiram um PTAS em que o número de máquinas faz parte da entrada do problema.

Considerando ainda o escalonamento em máquinas idênticas com objetivo de minimizar o *makespan*, mas agora focando no cenário de computação *online*, no qual não se conhece todas as tarefas a *priori*, Faigle et al. [1989] mostraram que a razão de competitividade de qualquer algoritmo online determinístico não pode ser menor que $(2 - \frac{1}{m})$, para $m = 2$ e $m = 3$. O já citado algoritmo *list scheduling* de Graham é $(2 - \frac{1}{m})$-competitivo, atingindo, portanto, a melhor garantia possível para aqueles valores de m. Em 1993, Galambos e Woeginger [1993] apresentaram um algoritmo $(2 - \frac{1}{m} - \epsilon_m)$-competitivo, com $\epsilon_m > 0$, mas tendendo à 0 conforme m cresce. O primeiro algoritmo com razão de competitividade assintoticamente menor que 2 foi apresentado por Bartal et al. [1992] e é $1{,}986$-competitivo. Karger et al. [1996] provaram um limitante superior $1{,}945$-competitivo e Albers [1999], em 1999, obteve um algoritmo $1{,}923$-competitivo. Em 2000, Fleischer e Wahl [2000] obtiveram um algoritmo $1{,}9201$-competitivo, a melhor razão atingida por um algoritmo determinístico para minimização de *makespan* num cenário online. Depois que Faigle et al. [1989] provaram um limitante inferior para estratégias determinísticas, quando $m = 2$ ou $m = 3$, houve tentativas de obter um limitante inferior para um número arbitrário de máquinas. O melhor conhecido foi obtido por Rudin III e Chandrasekaran [2003], que provaram que nenhum algoritmo online determinístico pode alcançar competitividade menor que uma razão de $1{,}88$.

No campo dos algoritmos aleatórios, Bartal et al. [1992] apresentaram um algoritmo probabilístico para minimização de *makespan* em duas máquinas, com razão de competitividade $4/3$, chamado *Rand-2*. Chen et al. [1994] e Sgall [1997] mostraram que nenhum algoritmo online probabilístico pode ter razão de competitividade inferior à $1/(1 - (1 - \frac{1}{m})^m)$, valor que tende a $e/(e-1) \approx 1{,}58$ quando m tende a infinito. Seiden [2000] mostrou um algoritmo probabilístico no qual a razão de competitividade é menor que a melhor razão conhecida para algoritmos determinísticos se $m \in \{3, \ldots, 7\}$. Albers [2002] desenvolveu um algoritmo probabilístico $1{,}916$-competitivo para todo m, o primeiro mais eficiente que um determinístico para m arbitrário.

Os algoritmos genéticos podem atuar como bons *frameworks* para o treinamento de redes neurais (Whitley et al. [1995]). Pesquisadores como Gupta et al. [2000] e Lang et al. [2020] usaram essa abordagem em variantes online do problema *Two-Stage Hybrid Flow Shop* obtendo resultados interessantes. Lattanzi et al. [2020] estudaram como técnicas de previsão via *machine learning* podem contribuir para o problema do escalonamento online. Essas técnicas consistem em prever as soluções parciais em relação a sazonalidade das demandas de tarefas. Ao estudar o problema *rent-or-buy*, Anand et al. [2020] afirmaram que a abordagem de utilizar *machine leaning* para otimização é uma rica direção para pesquisas futuras, tanto em algoritmos online ou em qualquer cenário onde os algoritmos pode adquirir vantagens de performance ao aprender padrões de entradas.

Contribuições. Neste trabalho projetamos um algoritmo para a versão online do problema do Escalonamento em Máquinas Identicas com Minimização de Makespan. O diferencial do nosso algoritmo é ele ser baseado em redes neurais, e a intuição por trás dessa abordagem é que, dada a situação de incerteza sobre as tarefas futuras do cenário online, um algoritmo que utiliza técnicas de aprendizado de máquina pode detectar padrões nas instâncias e assim obter escalonamentos

melhores. Nós comparamos nosso algoritmo em dois conjuntos de instâncias com o algoritmo List Scheduling, cujas soluções apresentam garantia de qualidade de pior caso.

Este trabalho tem a seguinte organização. Na Seção 2 definimos mais formalmente nosso objeto de estudo, apresentamos uma formulação em programação linear inteira para o problema, e introduzimos algoritmos online e análise competitiva, focando no algoritmo List Scheduling. Na Seção 3 detalhamos o funcionamento e treinamento do nosso algoritmo baseado em redes neurais. Na Seção 4 apresentamos os resultados computacionais. Finalmente, na Seção 5 fazemos nossas considerações finais e apontamos direções para trabalhos futuros.

2. Fundamentação Teórica

O Problema do Escalonamento em que estamos interessados considera um número m de máquinas idênticas e um conjunto com n tarefas, cada uma com seu respectivo tempo de processamento. A seguir ele é definido formalmente.

Definição 2.1 (Problema do Escalonamento em Máquinas Idênticas com Minimização de Makespan)**.** *Seja m o número de máquinas e n o número de tarefas, sendo que cada tarefa $i = 1, \ldots, n$ possui um tempo de processamento p_i. Em uma solução para esse problema, cada tarefa deve ser alocada em exatamente uma máquina. Seja l_i a carga na máquina i, para $i = 1, \ldots, m$. A carga l_i corresponde à soma dos tempos de processamento de todas as tarefas alocadas na máquina i. O objetivo é encontrar uma solução que minimiza $\max_{i=1,\ldots,m}\{l_i\} = t_{max}$, isto é, o makespan.*

Note que o makespan é tanto a carga da máquina mais carregada quanto o tempo de término do processamento da tarefa que encerra por último. Considere a variável de decisão x_{ij} descrita a seguir.

$$x_{ij} = \begin{cases} 1 & \text{se a tarefa } i \text{ é alocada na máquina } j, \\ 0 & \text{caso contrário,} \end{cases} \qquad \forall j \in [1, m], i \in [1, n].$$

Usando essas variáveis, modelamos o problema no seguinte Programa Linear Inteiro.

$$\begin{aligned} &\text{minimizar } t_{\max} \\ &\text{sujeito à } \sum_{j=1}^{m} x_{ij} = 1 && \text{para } i \in [1, n], \\ &\qquad \sum_{i=1}^{n} p_i x_{ij} \leq t_{\max} && \text{para } j \in [1, m], \\ &\qquad x_{ij} \in \{0,1\} && \text{para } j \in [1, m] \text{ e } i \in [1, n], \end{aligned}$$

sendo que as primeiras restrições garantem que cada tarefa seja alocada em uma máquina, e as seguintes garantem que o valor da função objetivo não é menor que a carga de cada máquina.

O Problema do Escalonamento com Minimização de Makespan é um problema NP-difícil [Garey e Johnson, 1975, 1978], o que significa que ele não pode ser solucionado em tempo polinomial, a não ser que P = NP. Além dessa dificuldade, problemas de escalonamento são frequentemente encontrados em sua versão online, na qual os dados são recebidos ao longo do tempo, e é necessário tomar decisões assim que cada tarefa chega, sem ter conhecimento das que estão por vir. Um algoritmo que opera neste cenário é um algoritmo online, e a qualidade de suas soluções pode ser analisada de forma semelhante aos de aproximação, comparando sua solução com a ótima do problema offline correspondente.

Definição 2.2 (Algoritmo Online Competitivo)**.** *Seja OPT(I) o valor ótimo para uma instância I de um problema de minimização em sua versão offline, ALG um algoritmo online e β uma constante independente da entrada. Um algoritmo é α-competitivo se, para qualquer instância I, devolve uma solução ALG(I) tal que*

$$ALG(I) \leq \alpha OPT(I) + \beta \ .$$

Na subseção seguinte apresentamos detalhes sobre o List Scheduling, que é um algoritmo competitivo clássico e um dos algoritmos que utilizamos nos testes computacionais deste trabalho.

2.1. List Scheduling

Dado um cenário com m máquinas idênticas funcionando em paralelo, temos uma sequência de tarefas $I = J_1, J_2, \ldots, J_n$ em que cada tarefa J_i tem um tempo de processamento p_i. As tarefas chegam uma por vez e a tarefa corrente deve ser alocada imediatamente, sem conhecimento das futuras. Preempção (interromper uma tarefa e retomá-la depois) não é permitida e o objetivo é minimizar o *makespan*. No artigo original sobre o *List Scheduling*, cujo pseudocódigo é apresentado no Algoritmo 1, Graham [1966] já investigava esse cenário. Para contrastar, note que

Algoritmo 1: List Scheduling

Entrada: O número de máquinas m e uma lista de tarefas, cada uma com tempo de processamento p_j.
Saída: Um escalonamento das tarefas nas máquinas.
1 **enquanto** *houver tarefas não escalonadas na lista de tarefas* **faça**
2 Escalone a próxima tarefa da lista na máquina menos carregada;

enquanto o *List Scheduling* é um algoritmo online, sua variante que precisa ordenar as tarefas não é, pois ela assume um conhecimento das tarefas futuras.

Teorema 2.1. *O algoritmo List Scheduling é $(2 - \frac{1}{m})$-competitivo.*

Demonstração. Para uma sequência arbitrária $I = J_1, \ldots, J_n$, considere o escalonamento construído pelo *List Scheduling* e seja t_{max} o *makespan* desse escalonamento. Sem perda de generalidade, renomeamos e ordenamos em ordem crescente de carga as m máquinas, de modo que a máquina 1 seja a menos carregada e que a carga da máquina m seja o *makespan*.

Considere um intervalo de tempo de comprimento t_{max}. A máquina m processa sem interrupções, as demais $m - 1$ máquinas processam um subconjunto das tarefas e depois, possivelmente, ficam um período de tempo ociosas. Durante seu tempo ativo, as máquinas processam $\sum_{i=1}^{n} p_i$ unidades. A Figura 1 representa um escalonamento produzido pelo algoritmo. Observemos a última tarefa J_k alocada na máquina m. Pela regra do *List Scheduling*, a máquina m tinha a menor carga no momento dessa atribuição. Portanto, a carga l_j de qualquer máquina $j = 1, \ldots, m - 1$ somada ao tempo de processamento p_k da tarefa J_k é um limitante superior para o makespan, i.e.,

$$t_{\max} \leq l_j + p_k \leq l_j \max_{i=1,\ldots,n} p_i \ .$$

Somando esta inequação ao longo das primeiras $m - 1$ máquinas e lembrando que a carga da

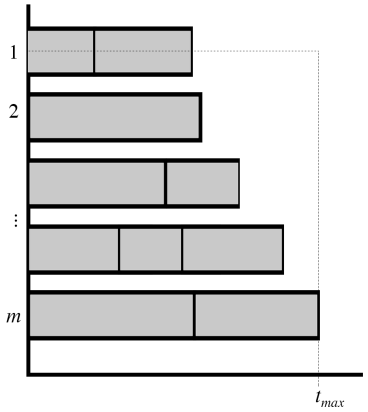

Figura 1: Escalonamento resultante do List Scheduling

máquina m é igual ao makespan, temos

$$\begin{aligned} mt_{\max} &\leq \sum_{j=1}^{m-1} l_j + (m-1) \max_{i=1,\ldots,n} p_i + l_m \\ &\leq \sum_{j=1}^{m} l_j + (m-1) \max_{i=1,\ldots,n} p_i \\ &\leq \sum_{i=1}^{n} p_i + (m-1) \max_{i=1,\ldots,n} p_i \ . \end{aligned}$$

Dividindo pelo número de máquinas temos o seguinte limitante superior para o makespan

$$t_{\max} \leq \frac{1}{m} \sum_{i=1}^{n} p_i + \left(1 - \frac{1}{m}\right) \max_{i=1,\ldots,n} p_i \ .$$

Seja $t^*_{\max}$ o makespan ótimo de um algoritmo offline, sabemos que $t^*_{\max} \geq \frac{1}{m} \sum_{i=1}^{n} p_i$ e que $t^*_{\max} \geq \max_{i=1,\ldots,n} p_i$, ou seja, $t^*_{\max}$ é limitado inferiormente tanto pela média dos tempos de processamento das tarefas, quanto pela tarefa mais longa. Dessa forma concluímos então que

$$t_{\max} \leq t^*_{\max} + \left(1 - \frac{1}{m}\right) t^*_{\max} = \left(2 - \frac{1}{m}\right) t^*_{\max} \ .$$

□

Faigle et al. [1989] mostraram que nenhum algoritmo online determinístico tem razão de competitividade menor que $2 - \frac{1}{m}$ para $m = 2$ e $m = 3$. Assim, nesses casos o *List Scheduling* tem a melhor garantia possível.

3. Algoritmo de Redes Neurais Competitivas

Nesta seção descrevemos nosso algoritmo para o problema do escalonamento, que é baseado na competição de redes neurais geradas aleatoriamente. Aquelas redes que tiverem melhores resultados na resolução do problema conseguem sobreviver para a próxima geração e gerar descendentes que sofrem pequenas mutações. Ao final de várias gerações, a melhor rede encontrada por

esse processo é utilizada na resolução dos problemas. Na Subseção 3.1 explicamos a arquitetura de redes neurais utilizada. Na Subseção 3.2 apresentamos o funcionamento do algoritmo para resolver o Problema Online do Escalonamento com uma rede já treinada. Na Subseção 3.3 apresentamos um pré-processamento realizado nas entradas. E finalmente, na Subseção 3.4 explicamos como se dá o treinamento do nosso algoritmo.

3.1. Arquitetura

Nesse trabalho utilizamos um modelo simples de Rede Neural Artificial baseado no trabalho de Wang [2003]. O modelo consiste em um grafo cujos vértices estão divididos em uma camada de entrada, algumas camadas escondidas e, no final, uma camada de saída, como é ilustrado na Figura 2. Na mesma, cada quadrado representa um número que compõe a entrada da rede

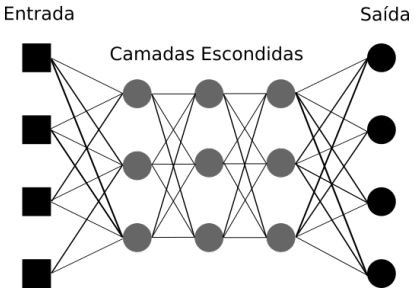

Figura 2: Um exemplo de arquitetura de Rede Neural contendo 4 entradas, 4 saídas e 3 camadas escondidas, com 3 neurônios por camada.

e cada círculo representa um neurônio. Os círculos pretos são os neurônios da camada de saída, cuja estrutura é idêntica a dos demais neurônios, porém cujos valores são tratados de modo distinto durante o uso da rede. A rede é dividida nas camadas $0, 1, \ldots, s$, em que a camada 0 é de entrada e s é a camada de saída. Um vértice (entrada ou neurônio) é identificado por uma dupla (k, i), em que k indica a camada do vértice e i indica a posição do mesmo na camada. Cada vértice de uma camada envia um valor para os vértices da camada seguinte. Assim, cada neurônio recebe um valor de cada vértice da camada anterior, como indicam as arestas (conexões) na figura, e a cada conexão está associado um valor chamado de peso. A saída de um neurônio i da camada k, com $1 \leq k \leq s$, é

$$h_i^k = \sigma\left(\sum_{j=1}^{N_{k-1}} V_{ij}^k h_j^{k-1} + b_i^k\right) \quad ,$$

sendo N_{k-1} o número de vértices na camada $k-1$, V_{ij}^k o peso que o neurônio (k,i) atribui à conexão de entrada j, e b_i^k o viés do neurônio (k,i). Note que, h_j^0 é um valor recebido pelo vértice j da camada de entrada. Por fim, σ é a função de ativação, cujo propósito é introduzir uma não-linearidade na rede neural. Na literatura encontramos funções que são utilizadas para essa finalidade, nesse trabalho optamos pelo uso da função ReLU (*Rectified Linear Unit*) que é definida por

$$ReLU(s) = \max\{0, s\}.$$

Dizemos que um neurônio foi **ativado** quando ele recebeu valores para suas conexões de entrada e produziu um valor na saída. Para ativar a rede realizamos a ativação dos neurônios separadamente, obedecendo a ordem das camadas. Primeiro ativamos todos os neurônios da primeira camada, depois os da segunda, e assim por diante até ativarmos a camada de saída. A saída de

cada neurônio da camada de saída será 0 ou um número positivo. Daremos para cada uma dessas saídas uma interpretação booleana. Se a saída for igual a 0 ela será interpretada como *false*. Caso contrário, ela será interpretada como *true*. Seja h_i^s o valor de saída do i-ésimo neurônio da camada de saída, para $i = 0, 1, \ldots, N_s$. Definimos como a saída da rede, ou seja, o resultado produzido pela rede, o valor i, que é o índice do primeiro neurônio da camada de saída cujo o valor h_i^s foi avaliado como *true*. Assim, o resultado da rede é i se, e somente se, $h_j^s = 0$ para $j < i$ e $h_i^s > 0$. Cada saída possível de uma rede, corresponde a uma ação específica que o algoritmo deverá executar.

Essa interpretação da rede neural nos oferece uma oportunidade de usá-la como um decisor. Desta forma um algoritmo alimenta a entrada da rede com dados do problema e a rede decide qual ação deve ser executada. Por exemplo, no Problema de Escalonamento, a rede pode decidir em qual máquina será alocada a próxima tarefa. Para tornar isso possível, precisamos encontrar uma rede cujas ações ordenadas em sequência resulta em uma solução competitiva. O processo de encontrar essa rede é chamado de treinamento.

3.2. Algoritmo de Decisão

Antes de abordarmos o treinamento da rede, vamos descrever o algoritmo que utiliza uma rede neural treinada para resolver o problema do escalonamento. Nesse algoritmo, cujo pseudocódigo é apresentado em Algoritmo 2, a rede será consultada uma vez para cada tarefa que chega, a fim de decidir em qual máquina a tarefa será processada. Seja $L = (l_1, l_2, l_3, ..., l_m)$ uma

Algoritmo 2: Escalonamento usando uma Rede Neural

Entrada: Uma sequência de tarefas I, um número de máquinas m e uma rede R
Saída: Um escalonamento das tarefas I em m máquinas

```
enquanto existem tarefas não alocadas faça
    Seja L uma configuração de carga das máquinas;
    Seja J a próxima tarefa a ser alocada;
    (L', J') = tratamento(L, J);
    i' = R(J', L');
    Aloque a tarefa J na i'-ésima máquina mais carregada;
```

configuração em que l_i é a carga da i-ésima máquina. Sendo J a próxima tarefa a ser alocada e considerando uma configuração L, temos que o par $(J', L') = \text{tratamento}(J, L)$ corresponde, respectivamente, a uma configuração e uma tarefa após a eliminação de algumas equivalências. Essas equivalências e o tratamento das mesmas serão explicados na subseção seguinte. Temos que $i' = R(J', L')$ é o resultado da rede R, que indica o índice i' da máquina na qual a tarefa J' deve ser alocada na configuração L'.

3.3. Tratamento das Entradas

Com o intuito de tornar mais eficiente o reconhecimento de padrões por parte da rede neural, identificamos três tipos de equivalência entre configurações de carga das máquinas. A seguir descrevemos essas equivalências e explicamos como eliminá-las, o que pode ser feito pela função de tratamento do algoritmo. Chamamos o primeiro tipo de equivalência por simetria. A Figura 3 mostra três configurações que apresentam essa equivalência, pois são idênticas a menos da ordem das máquinas. Para eliminar essa equivalência, dada uma configuração L, basta que suas máquinas sejam ordenadas (e renomeadas) em ordem crescente de cargas, gerando uma configuração L'. Note que, dada uma configuração com m máquinas com cargas distintas, existem $m!$ configurações

Figura 3: Um exemplo de configurações que apresentam equivalência por simetria.

equivalentes. Isso indica quão grande pode ser o impacto da eliminação dessa equivalência no espaço de configurações que a rede deve analisar.

O segundo tipo chamamos de equivalência por escala. A Figura 4 mostra três configurações em que a carga relativa de cada máquina em função do tempo de processamento da tarefa atual é a mesma. Mais precisamente, a razão entre a carga da i-ésima máquina e o tempo

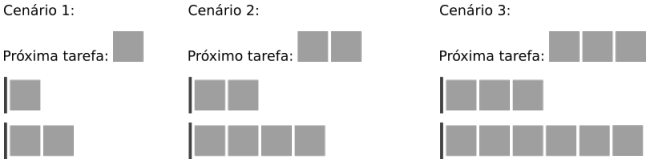

Figura 4: Um exemplo de configurações que apresentam equivalência por escala.

de processamento da tarefa atual é a mesma nas três configurações, para $i = 1, \ldots, m$. Assim o impacto da alocação da nova tarefa no balanceamento relativo de cada configuração é o mesmo. Para eliminar essa equivalência, dada uma configuração L e uma tarefa J, basta dividir a carga de cada máquina em L pelo tempo de processamento da tarefa J. Quando essa normalização é adotada o tamanho da tarefa atual passa a ser unitário. Note que, o número de equivalências deste tipo é infinito, dependendo apenas da variação do tamanho da próxima tarefa.

O terceiro e último tipo de equivalência que identificamos é chamado de equivalência por carga. A Figura 5 mostra três configurações que apresentam essa equivalência. Tomando as

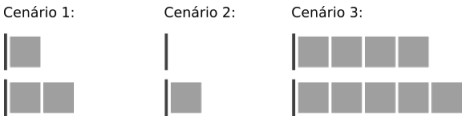

Figura 5: Um exemplo de configurações que apresentam equivalência por carga.

configurações da figura como exemplo, observem que embora as cargas das máquinas sejam distintas nos três cenários, o perfil da distribuição de cargas é o mesmo. Ou seja, a carga da máquina mais carregada é 1 unidade maior que da máquina menos carregada nos três cenários. Como desejamos minimizar o valor absoluto do *makespan*, podemos inferir que a decisão da rede não deveria ser diferente nesses três cenários, independente do total de cargas passadas. Para eliminar essa equivalência, basta subtrair o menor valor da menor carga da carga de cada máquina.

Qualquer combinação dos três procedimentos para eliminação de equivalência descritos anteriormente pode ser usada na função de tratamento do algoritmo. No entanto, se o primeiro tipo for usado é essencial lembrar de converter o resultado da rede de i' para o índice i correspondente antes da reordenação das máquinas. Também é interessante perceber que a ordem em que os procedimentos de eliminação de equivalência são aplicados não afeta o resultado. Isso porque o primeiro

só altera a ordem das máquinas, enquanto o segundo e terceiro não mudam a ordem relativa de carga das máquinas. Além disso, o segundo procedimento só depende do tempo de processamento da tarefa atual, enquanto o terceiro procedimento só depende da carga que é comum a todas as máquinas, cujo valor absoluto pode ser alterado pelo procedimento 2, mas de modo homogêneo ao longo das máquinas.

3.4. Algoritmo de Treinamento

O treinamento da Rede Neural no contexto de redes competitivas objetiva encontrar um conjunto de pesos e vieses para uma topologia de rede previamente escolhida, de forma que, ao usar essa rede como decisora, o *makespan* obtido seja pequeno. Para avaliar uma rede durante o treinamento usamos o somatório dos *makespans* obtidos por ela ao longo de todas instâncias de treinamento. Chamamos esse somatório de *fitness*. Para a etapa de treinamento utilizamos um algoritmo genético, cujo pseudocódigo é apresentado no Algoritmo 3.

Algoritmo 3: Treinamento de Rede Neural para o problema do escalonamento online

Entrada: Um conjunto de instâncias de treinamento, uma topologia de rede neural
Saída: A rede com melhor fitness que for encontrada

```
1  i = 1;
2  Crie a primeira geração com p redes aleatórias;
3  c = ⌊vp⌋;
4  enquanto i ≤ g E j ≤ número máximo de gerações sem melhoria faça
5      Calcule o fitness de todas as redes da geração i;
6      se a melhor rede da geração i superar a melhor rede global então
7          atualize a melhor rede global;
8          j = 0;
9      Ordene as redes pelo fitness;
10     Copie as c melhores redes para a próxima geração;
11     d = p mediana(0.1, i/g, 0.9);
12     Crie d redes descendentes para a próxima geração;
13     a = p − c − d;
14     Crie a redes aleatórias;
15     j = j + 1;
16     i = i + 1;
17 Devolva a melhor rede;
```

Considere os seguintes parâmetros do algoritmo de treinamento: g é o número total de gerações, p é o número de indivíduos da população e v é a taxa de elitização, i.e., a fração da população que será elite. Na primeira iteração do algoritmo todos os p indivíduos são redes geradas com pesos e vieses escolhidos aleatoriamente. Definimos o tamanho do conjunto de indivíduos elite como $c = \lfloor vp \rfloor$. No início da iteração avaliamos o *fitness* de cada rede daquela geração. Em seguida, verificamos se foi encontrada uma rede com *fitness* melhor que o da melhor rede global até o momento. Se for o caso, atualizamos a melhor rede global e zeramos o contador j de iterações sem melhoria. Em seguida, ordenamos as redes pelo *fitness* e copiamos as c melhores redes para a população da próxima geração, formando o conjunto elite. Então, calculamos o número d de descendentes dos elites, que são cópias das redes do conjunto elite que sofrem alguma mutação. Uma mutação consiste em alterar cada peso e viés da rede, somando a este um número aleatório

pequeno. Note que d cresce ao longo das iterações, sendo limitado a no mínimo 10% do tamanho p da população e a no máximo 90% de p. Por fim, completamos a população da próxima geração com a indivíduos gerados aleatoriamente.

Ao longo das iterações o valor d vai crescendo e o valor a vai diminuindo. A intuição por trás dessa decisão de projeto é que, no início a chance de uma rede boa surgir aleatoriamente é maior. Com o passar das gerações, temos diversas redes boas no conjunto elite. Assim, passa a ser mais provável obter uma rede melhor a partir de um descendente de uma dessas redes, do que de uma rede completamente aleatória. A topologia de rede citada no Algoritmo 3, é composta pelo seguinte conjunto de parâmetros $\{$número de entrada, número de camadas escondidas, número de neurônios em cada camada escondida, número de saídas$\}$.

4. Resultados Experimentais

Nesta seção apresentamos os resultados computacionais nos quais comparamos o desempenho do nosso algoritmo com o do List Scheduling em alguns conjuntos de instâncias. Para analisar a capacidade das redes neurais reconhecerem padrões e executar heurísticas aprendidas no treinamento, para um perfil de instâncias com características em comum, definimos dois grupos de instâncias, um de instâncias homogêneas e outro de instâncias heterogêneas. Cada instância é composta por 100 tarefas, nas instâncias homogêneas as tarefas possuem tamanhos inteiros sorteados uniformemente no intervalo $[10, 50]$, já as instâncias heterogêneas terão 90 tarefas com tamanhos inteiros sorteados uniformemente no intervalo $[10, 50]$ e 10 tarefas *grandes*, com tamanhos sorteados uniformemente no intervalo $[100, 150]$. Para cada perfil de instância foram geradas 100 instâncias de treinamento e 100 instâncias de teste. Os algoritmos foram implementados em C++ e executados numa máquina i3-6006U CPU @ 2.00GHz, 4 Giga.

Foram treinadas redes distintas para as instâncias homogêneas e para as instâncias heterogêneas. Treinamos ao todo 4 topologias de redes diferentes, utilizando dois subconjuntos de tratamentos de entradas citados na subseção 3.3, o primeiro considera todos os tratamentos, essa configuração identificamos como ORS, e o segundo, que identificamos como OR, não executamos o tratamento que elimina as equivalências por carga, totalizando 16 redes treinadas. Todas as instâncias usadas nesse trabalho possuem 10 máquinas, então as redes possuem 10 entradas (com as cargas das máquinas) e 10 saídas (indicando em qual máquina será escalonada a tarefa). As 4 topologias escolhidas, possuem número de camadas escondidas e números de neurônios escondidos: 2x2, 4x4, 8x8, 10x10. Os valores de *makespan* médios reportados, são o resultado da execução dessas redes no conjunto de instâncias de teste. Foi definido o limite de 200 gerações sem melhorias e os resultados se encontram nas Tabelas 1 e 2.

	Homogêneas ORS				Homogêneas OR			
	Treino (s)	Ger	Exec (s)	Makespan	Treino (s)	Ger	Exec (s)	Makespan
2x2	105.985	248	5.82E-05	458.33	161.39	308	5.37E-05	473.68
4x4	293.035	508	6.95E-05	595.16	160.066	238	6.66E-05	458.41
8x8	694.926	502	0.0002696	457.64	642.685	352	0.00018352	457.49
10x10	508.311	218	0.00024798	456.04	1303.59	449	0.00025214	460.93

Tabela 1: Médias dos tempos de treinamento (Treino), números de gerações (Ger), tempos de execução (Exec), e *Makespan* nos testes de nossas redes com instâncias homogêneas.

O algoritmo *List Scheduling* foi também executado nas instâncias de teste de ambos os grupos. Nas 100 instâncias de teste homogêneas, o algoritmo *List Scheduling* obteve um *makespan* médio igual a 330.77 e no grupo de instâncias heterogêneas 468.49. Nas instâncias do grupo homogêneo, a rede que obteve o menor *makespan* médio foi a rede com 10 camadas escondidas e 10

	Heterogêneas ORS				Heterogêneas OR			
	Treino (s)	Ger	Exec (s)	Makespan	Treino (s)	Ger	Exec (s)	Makespan
2x2	94.3669	223	5.54E-05	457.82	272.543	665	5.20E-05	458.8
4x4	218.941	378	8.89E-05	454.33	310.167	432	6.95E-05	455.02
8x8	864.843	596	0.00015846	454.29	799.094	552	0.00015913	458.74
10x10	2488.73	1028	0.00024773	**447.89**	2622.9	1070	0.00024797	**447.89**

Tabela 2: Médias dos tempos de treinamento (Treino), números de gerações (Ger), tempos de execução (Exec), e *Makespan* nos testes de nossas redes com instâncias heterogêneas.

neurônios por camada, com configuração ORS. O *makespan* médio foi de 456.04 que não é menor do que foi obtido pelo *List Scheduling*. Já no grupo heterogêneo, o menor *makespan* médio foi obtido por duas redes com 10 camadas escondidas e 10 neurônios por camada, uma com configuração ORS e outra com configuração OR. Nesse caso o menor *makespan* médio foi de 447.89 que é menor do que o do *List Scheduling*. Observamos que tanto nas instâncias heterogêneas quanto nas homogêneas o tratamento que elimina a equivalência por carga contribuiu apenas marginalmente. Conjecturamos que isso ocorre pois esse tratamento elimina informações sobre o tamanho total da instância. Na Tabela 3 comparamos os *makespans* médios do algoritmo *List* com as melhores redes obtidas nos treinamentos. Nas instâncias homogêneas, nenhuma rede obteve um *makespan* menor que o *List*, já nas instâncias heterogêneas, todas as redes treinadas nesse grupo de instâncias obtiveram *makespans* menores que o *List*, sendo as duas redes de tamanhos 10x10 as melhores delas, ambas com o *makespan* de 447.89. Nas instâncias homogêneas a rede neural não obteve nenhuma solução melhor que o *List* enquanto nas instâncias heterogêneas foram obtidas soluções melhores em 81% das instâncias de teste.

	List		RNC (ORS)	
Instância	Tempo	Média Makespan	Tempo	Média Makespan
Homogênea	9.92E-06	**330.77**	0.00024798	456.04
Heterogênea	8.94E-06	468.49	0.00024773	**447.89**

Tabela 3: Comparação do método List com as melhores redes que obtivemos para cada tipo de instância. Tanto para as instâncias homogêneas quanto para as instâncias heterogêneas, selecionamos as redes com 10 camadas escondidas e 10 neurônios escondidos, com todos os tratamentos de entrada descritos na Seção 3.3.

5. Conclusões

Neste trabalho usamos redes neurais para resolver o problema de escalonamento online, sendo que essas redes são treinadas usando um algoritmo genético. Nossos resultados mostram que a rede treinada para tratar um perfil de instância heterogêneo em relação aos tamanhos das tarefas obteve melhores resultados que o algoritmo clássico *List Scheduling*. Nossa hipótese é que as redes neurais conseguem detectar padrões de entrada, o que lhes permite tomar boas decisões de alocação frente a incerteza das tarefas vindouras. Isso nos motiva a testar essa abordagem baseada em redes neurais para outros perfis de instâncias.

Em trabalhos futuros pretendemos explorar diferentes topologias de rede, algoritmos de treinamento e regras de mutação. Além disso, será importante testar a robustez do método no problema do escalonamento online com número de máquinas variável. Também pretendemos atacar outros problemas de otimização, online e offline, utilizando redes neurais competitivas.

Além disso, elaboramos um conjunto de instâncias de treinamento e teste que estão disponíveis no endereço `https://hokama.com.br/online_scheduling/`.

Referências

Albers, S. (1999). Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473.

Albers, S. (2002). On randomized online scheduling. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, p. 134–143.

Anand, K., Ge, R., e Panigrahi, D. (2020). Customizing ml predictions for online algorithms. In *International Conference on Machine Learning*, p. 303–313. PMLR.

Bartal, Y., Fiat, A., Karloff, H., e Vohra, R. (1992). New algorithms for an ancient scheduling problem. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, p. 51–58.

Chen, B., van Vliet, A., e Woeginger, G. J. (1994). A lower bound for randomized on-line scheduling algorithms. *Information Processing Letters*, 51(5):219–222.

Faigle, U., Kern, W., e Turán, G. (1989). On the performance of on-line algorithms for partition problems. *Acta cybernetica*, 9(2):107–119.

Finn, G. e Horowitz, E. (1979). A linear time approximation algorithm for multiprocessor scheduling. *BIT Numerical Mathematics*, 19(3):312–320.

Fleischer, R. e Wahl, M. (2000). On-line scheduling revisited. *Journal of Scheduling*, 3(6):343–353.

Galambos, G. e Woeginger, G. J. (1993). An on-line scheduling heuristic with better worst-case ratio than graham's list scheduling. *SIAM Journal on Computing*, 22(2):349–355.

Garey, M. R. e Johnson, D. S. (1975). Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411.

Garey, M. R. e Johnson, D. S. (1978). "strong"np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508.

Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581.

Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429.

Gupta, J. N., Sexton, R. S., e Tunc, E. A. (2000). Selecting scheduling heuristics using neural networks. *INFORMS Journal on Computing*, 12(2):150–162.

Hochbaum, D. S. e Shmoys, D. B. (1987). Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162.

Karger, D. R., Phillips, S. J., e Torng, E. (1996). A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20(2):400–430.

Lang, S., Reggelin, T., Behrendt, F., e Nahhas, A. (2020). Evolving neural networks to solve a two-stage hybrid flow shop scheduling problem with family setup times. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.

Lattanzi, S., Lavastida, T., Moseley, B., e Vassilvitskii, S. (2020). Online scheduling via learned weights. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 1859–1877. SIAM.

Rudin III, J. F. e Chandrasekaran, R. (2003). Improved bounds for the online scheduling problem. *SIAM Journal on Computing*, 32(3):717–735.

Seiden, S. S. (2000). Online randomized multiprocessor scheduling. *Algorithmica*, 28(2):173–216.

Sgall, J. (1997). A lower bound for randomized on-line multiprocessor scheduling. *Information Processing Letters*, 63(1):51–55.

Wang, S.-C. (2003). Artificial neural network. In *Interdisciplinary computing in java programming*, p. 81–100. Springer.

Whitley, D. et al. (1995). Genetic algorithms and neural networks. *Genetic algorithms in engineering and computer science*, 3:203–216.