Universidade Estadual de Londrina

PEDRO ZAFFALON DA SIVA

ALOCAÇÃO DE REGISTRADORES UTILIZANDO MACHINE LEARNING

LONDRINA
2023

PEDRO ZAFFALON DA SIVA

ALOCAÇÃO DE REGISTRADORES UTILIZANDO MACHINE LEARNING

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

Coorientador: Profa. Dra. Helen Cristina de Mattos Senefonte

LONDRINA

2023

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

ZAFFALON DA SILVA, PEDRO .

Alocação de Registradores utilizando Machine Learning / PEDRO ZAFFALON DA SILVA. - Londrina, 2023.

75 f. : il.

Orientador: Wesley Attrot.

Coorientador: Helen Cristina de Mattos Senefonte.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2023.

Inclui bibliografia.

1. Estudo sobre a aplicação de machine learning na alocação de registradores. - TCC. I. Attrot, Wesley. II. Cristina de Mattos Senefonte, Helen. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. IV. Título.

CDU 519

PEDRO ZAFFALON DA SIVA

ALOCAÇÃO DE REGISTRADORES UTILIZANDO MACHINE LEARNING

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina

Profa. Dra. Helen Cristina de Mattos Senefonte
Universidade Estadual de Londrina – UEL

Prof. Vanessa Matias Leite
Universidade Estadual de Londrina – UEL

Londrina, 31 de maio de 2023.

AGRADECIMENTOS

Primeiramente, agradeço ao meu professor orientador Wesley Attrot pelo auxílio e suporte no desenvolvimento desse trabalho.

Também gostaria de agradecer aos demais professores presentes no Departamento de Computação da Universidade Estadual de Londrina, especialmente a professora Neyva Maria Lopes Romeiro, que me orientou e aconselhou durante toda a graduação.

Agradeço aos meus colegas de curso, que me acompanharam durante minha jornada acadêmica. Em especial, agradeço aos meus amigos mais próximos, Blenda e Rafael, que me ajudaram a enfrentar os desafios e me apoiaram nos tempos difíceis.

Por fim, gostaria de agradecer aos meus pais e irmãos, que sempre me apoiaram e incentivaram.

SILVA, P. Z.. **Alocação de Registradores utilizando Machine Learning**. 2023. 75f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

A alocação de registradores é uma etapa com grande impacto na otimização de códigos gerados pelo compilador. Geralmente, sua resolução é realizada através da coloração de grafo, um problema NP-completo. Devido a sua importância, várias heurísticas foram propostas para a sua resolução. Contudo, a criação delas é um processo complexo e altamente especializado. Em um contexto no qual machine learning é cada vez mais aplicado em otimizações de compiladores, sua utilização para melhorar a alocação de registradores se torna uma opção interessante. Porém, devido à necessidade de corretude, não presente em outras formas de otimização realizadas por compiladores, apenas recentemente esse tema foi mais pesquisado. Desta forma, este trabalho propõe um estudo sobre a utilização de Machine Learning na alocação de registradores, com objetivos de informar técnicas desenvolvidas e apontar caminhos promissores na área.

Palavras-chave: Alocação de Registradores, Machine Learning, Otimização de compiladores

SILVA, P. Z.. **Register Allocation with Machine Learning**. 2023. 75p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

Register allocation is a important phase for compiler optimization, generally mapped to the graph coloring, which is a NP-complete problem. Because of its impact for quality code generation, various heuristic algorithms have been proposed. However, heuristics development is a complex process and requires very specialized domain expertise. Recently, several Machine Learning based approaches are being proposed to solve compiler optimization problems. Still, due to the challenge of ensuring correctness, few works applied Machine Learning to register allocation. This works aims to contribute to the research of Machine Lerning usage to optimize register allocation by studying and describing methods present in the literature and promissing directions for future works.

Keywords: Register Allocation, Machine Learning, Compiler Optimization

LISTA DE ILUSTRAÇÕES

Figura 1 – Alocador estilo Chaitin. Essa figura foi retirada do trabalho de Brigss *et al.* [1]. 15
Figura 2 – Alocador estilo Briggs. Esta figura foi retirada do trabalho de Brigss *et al.* de [1]. 17
Figura 3 – Exemplo de grafo 3-colorável com vértices com 3 arestas. 17
Figura 4 – Grafo de Interferência do bloco. 18
Figura 5 – Remoção do vértices do grafo. 19
Figura 6 – Remoção dos vértices do grafo. 19
Figura 7 – Tempos de vida do exemplo de Linear Scan. 22
Figura 8 – Exemplo de coloração de grafos com PBQP. Essa figura foi retirada do trabalho de Buchwald *et al.* [2]. 23
Figura 9 – Exemplo de remoção RE no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2]. 24
Figura 10 – Exemplo de remoção R1 no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2]. 24
Figura 11 – Exemplo de remoção R2 no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2]. 24
Figura 12 – Exemplo de remoção RN no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2]. 25
Figura 13 – Grafo PBQP do exemplo. 26
Figura 14 – Grafo PBQP do exemplo após a remoção de *a*. 27
Figura 15 – Grafo PBQP do exemplo após a remoção de *b*. 27
Figura 16 – Grafo PBQP do exemplo após a remoção de *c*. 28
Figura 17 – Grafo PBQP do exemplo após a remoção de *d*. 28
Figura 18 – Ilustração de região de referência. Esta figura foi retirada do trabalho de Bergner *et al.* [3]. 43
Figura 19 – Ilustração de *live range splitting*. 45
Figura 20 – Exemplo de *containment graphs*. Esta figura foi retirada do trabalho de Cooper *et al.* [4]. 46
Figura 21 – *Containment graphs* do exemplo. 47
Figura 22 – Inteligência artificial e suas subáreas. 49
Figura 23 – Exemplo de uma rede neural multicamadas. 52
Figura 24 – Operações utilizadas em um neurônio. 53
Figura 25 – Exemplo de uma célula LSTM. Esta figura foi retirada do trabalho de Menezes dos Anjos. [5]. 55

Figura 26 – Exemplo do funcionamento de uma RNN/LSTM. Esta figura foi retirada do trabalho de Das *et al.* [6]. 55
Figura 27 – Ilustração do modelo de Das *et al.* [6]. 56
Figura 28 – Ilustração simplificada do funcionamento do alocador de VenkataKeerthy *et al.* [7]. 58
Figura 29 – Ilustração do funcionamento do Monte Carlo *tree search*. Esta figura foi retirada do trabalho de Chaslot *et al.* [8]. 60
Figura 30 – Ilustração simplificada do modelo de Kim *et al.* [9]. 61

LISTA DE TABELAS

Tabela 1 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura x86_64, usando os alocadores Chaitin-Briggs e os alocadores do LLVM. Esta tabela foi retirada da dissertação de mestrado de Lopes da Silva [10]. 29
Tabela 2 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura ARM Cortex-A8, usando os alocadores Chaitin-Briggs e os alocadores do LLVM. Esta tabela foi retirada da dissertação de mestrado de Lopes da Silva [10]. 30

SUMÁRIO

1 INTRODUÇÃO 12

2 TÉCNICAS TRADICIONAIS DE ALOCAÇÃO DE REGISTRADORES 14

2.1 Coloração de Grafos 14

2.1.1 Alocador de Chaitin 15

2.1.2 Alocador de Briggs 16

2.1.3 Exemplo com o Alocador de Briggs 17

2.2 Linear Scan 20

2.2.1 Exemplo de Linear Scan 21

2.3 Partitioned Boolean quadratic programming (PBQP) 23

2.3.1 Exemplo de PBQP 26

2.4 Comparação entre as Abordagens de Alocação Global 28

2.5 Alocação de Registradores Local 31

2.5.1 Exemplo de Alocação de Registradores Local 32

2.5.2 Diferenças entre Alocação Local e Global 34

3 MINIMIZAÇÃO DE SPILL 35

3.1 Heurísticas de Chaitin 35

3.2 Heurísticas de Bernstein 37

3.3 Rematerialization 38

3.3.1 Exemplo de Rematerialization 40

3.4 Spilling por Região de Interferência 42

3.4.1 Exemplo de Spilling por Região de Interferência 43

3.5 Live Range Splitting 45

3.5.1 Exemplo de Live Range Splitting 47

4 MACHINE LEARNING 49

4.1 Tipos de Aprendizado 50

4.1.1 Reinforcement learning 50

4.2 Redes Neurais 51

4.3 Deep Learning 53

5 APLICAÇÃO DE MACHINE LEARNING NA ALOCAÇÃO DE REGISTRADORES 54

5.1 Algoritmo de coloração de grafo aproximada baseada em deep learning 54

5.2 Alocação de registradores com Reinforcement Learning 57
5.3 Alocação de registradores com PBQP com deep reinforcement learning 59
5.4 Trabalhos relacionados 61

6 DIREÇÕES PROMISSORAS PARA TRABALHOS FUTUROS 63
6.1 Coloração de grafos 63
6.2 PBQP 64
6.3 Minimização de spill 65
6.4 Datasets 65
6.5 Heurísticas para decisões de spill 66

7 CONCLUSÃO 68

REFERÊNCIAS 69

1 INTRODUÇÃO

Alocação de registradores é a etapa do compilador responsável pelo mapeamento das variáveis para o armazenamento físico do computador, decidindo se cada variável será salva em um registrador ou na memória principal. É importante considerar a grande diferença entre a velocidade de acesso dos registradores e da memória principal. Por causa disso é necessário minimizar o armazenamento de variáveis na memória para que códigos gerados por compiladores obtenham melhor desempenho [11].

Devido a sua importância para a qualidade da geração de código, a alocação de registradores é um problema que é extensamente pesquisado há décadas. Geralmente é abstraído para a coloração de grafos, um problema de otimização combinatória NP-completo [12, 13, 14], onde cada vértice representa o tempo de vida de uma variável, de forma que uma aresta indica que as variáveis precisam ser armazenadas simultaneamente em algum momento. Desta forma, sendo as cores os registradores que serão alocados, é necessário atribuir cores sem que vértices ligados por uma aresta apresentem a mesma cor.

Para resolver este problema, várias heurísticas foram propostas durante os últimos 40 anos [14, 15, 1] e novas continuam sendo pesquisadas [16]. Porém, o desenvolvimento de heurísticas é um processo complexo, exigindo especialidade em campos bastante específicos, tanto em construção de compiladores, quanto em arquitetura de hardware [17]. Além disso, as heurísticas utilizadas apresentam desempenhos que podem ser melhorados em termos de otimização e muitas vezes precisam ser específicas para as arquiteturas [7].

Por outro lado, com o aumento da utilização de *Machine Learning* [18, 19] em diversas áreas, abordagens baseadas em redes neurais foram propostas para melhorar as otimizações realizadas pelos compiladores. Geralmente essas abordagens são aplicadas em problemas como *phase ordering* [20, 21], *throughput prediction* [22], ou na criação de heurísticas para otimizações [23, 24, 17]. Observa-se que nos processos citados não é necessário garantir a corretude, visto que erros afetariam apenas o desempenho do código compilado, mas não o seu funcionamento. Devido a maior complexidade, pouco foi explorado em problemas contendo restrições semânticas, como a alocação de registradores, na qual soluções incorretas podem implicar na perda de valores salvos em variáveis, por exemplo. Desta forma, apenas recentemente abordagens baseadas em *Machine Learning* foram aplicadas para resolução da alocação de registradores [6, 7, 25, 9, 26], as quais obtiveram resultados promissores, apesar da pequena quantidade de pesquisas envolvendo esse tema.

Neste contexto, é proposto neste trabalho um estudo sobre as principais técnicas e

resultados presentes na literatura sobre a utilização de *Machine Learning* para otimização da alocação de registradores. Sendo assim, por meio deste trabalho, será possível aprender sobre os métodos que já foram explorados nessa área, além de encontrar caminhos promissores que podem ser abordados em futuras pesquisas.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 descreve as principais abordagens e algoritmos utilizados por compiladores para realizar a alocação de registradores; o Capítulo 3 apresenta técnicas empregadas para minimizar o custo de *spill*; o Capítulo 4 consiste em uma fundamentação teórica sobre as principais técnicas de *machine learning*; o Capítulo 5 apresenta os principais trabalhos sobre aplicação de *machine learning* na alocação de registradores presentes na literatura; o Capítulo 6 aponta direções promissoras para futuras pesquisas sobre o tema; e por fim o Capítulo 7 apresenta a conclusão do trabalho.

2 TÉCNICAS TRADICIONAIS DE ALOCAÇÃO DE REGISTRADORES

Devido a sua localização na CPU (*Central Processing Unit*), registradores são as unidades mais rápidas da hierarquia de memória. Frequentemente, são as únicas localizações que podem ser acessadas diretamente pela maioria das operações [27]. Por outro lado, operações com memória são custosas energeticamente e ineficientes em comparação com registradores. Um único acesso à memória envolve diversos ciclos de instruções, enquanto dois registradores podem ser lidos e um escrito usando apenas um ciclo de instrução [28, 10].

Portanto, as variáveis de códigos devem ser preferencialmente armazenadas em registradores. No entanto, a quantidade de registradores é bastante limitada. Por exemplo, processadores 32-bit x86 apresentam apenas oito registradores de uso geral, processadores 16-bit x86 contêm 16, e processadores ARM e PowerPC contêm apenas 32 registradores de tipo inteiro [11]. Sendo assim, é impossível mapear todos os valores em registradores na maioria dos casos. Nessas situações, é preciso enviar algumas variáveis para a memória, recarregando o seu valor para um registrador apenas antes de seus usos. Esse processo é chamado de *spill*.

Desta forma, é necessário utilizar métodos eficientes de gerenciamento de registradores para mapear as variáveis da melhor forma possível. A qualidade de um alocador afeta diretamente o desempenho de códigos gerados por compiladores. Porém, a alocação de registradores é um problema NP-completo [12, 13, 14], então é impossível criar um alocador que obtém soluções ótimas para todos os casos.

Neste contexto, diversos algoritmos foram propostos para realizar a alocação de registradores, além de melhorias de métodos já existentes. Dependendo da arquitetura ou das características dos programas que são compilados, certos métodos podem ser mais adequados que outros. Assim, é importante conhecer as vantagens e desvantagens de cada abordagem ao escolher ou desenvolver um alocador. As próximas seções apresentam as principais técnicas desenvolvidas.

2.1 Coloração de Grafos

A abstração mais utilizada para a alocação de registradores é o grafo de interferência [11]. Os vértices são os tempos de vida das variáveis e as arestas as interferências entre os tempos de vida, sendo o grau do vértice o seu número de arestas. Assim, a alocação de registradores é reduzida ao problema de coloração de grafo, onde as cores representam os registradores físicos do processador [14].

Um grafo é k-colorável se é possível atribuir para cada vértice uma das k cores sem que vértices ligados por uma aresta tenham a mesma cor, sendo esse o principal objetivo da alocação de registradores. Se o grafo não for k-colorável, é necessário realizar o processo de *spill*, que consiste no uso da memória para armazenar a variável cujo tempo de vida é representada pelo vértice. Caso não seja possível evitar *spills*, o objetivo passa a ser minimizar o seu custo [14].

2.1.1 Alocador de Chaitin

Chaitin *et al.* [14] foram os primeiros a implementar um alocador baseado em coloração de grafo. A Figura 1 ilustra um alocador no estilo de Chaitin, sendo dividido em sete fases:

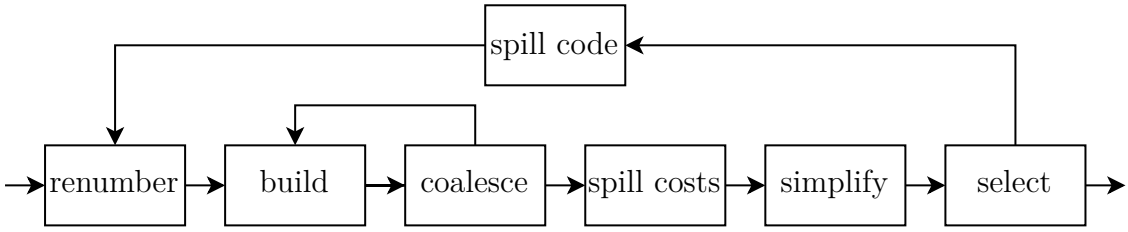


Figura 1 – Alocador estilo Chaitin. Essa figura foi retirada do trabalho de Brigss *et al.* [1].

- *Renumber* obtêm os tempos de vida das variáveis do programa.
- *Build* constrói o grafo de interferência.
- *Coalesce* tenta diminuir o número de vértices. Dois tempos de vida são combinados se a definição inicial de uma variável é uma cópia da outra e eles não interferem entre si. A instrução de cópia é eliminada e um novo tempo de vida é criado combinando as duas variáveis. Caso esta fase modifique o grafo, as etapas *build* e *coalesce* são realizadas novamente.
- *Spill costs* estima, para cada tempo de vida, o custo temporal das instruções que seriam adicionadas caso a variável sofresse *spill*. O custo é estimado a partir do número de operações *load* e *store* necessárias caso a variável fosse armazenada na memória.
- *Simplify* define uma ordem de prioridade dos tempos de vida. Cria uma pilha vazia e repete os seguintes passos até que o grafo esteja vazio:
  - Caso exista um vértice cujo grau (número de arestas) é menor que o número de cores, o vértice e todas as suas arestas são removidas do grafo. O tempo de vida é adicionado na pilha.

  - Caso contrário, escolha um vértice para *spill*. O vértice e suas arestas são removidas do grafo e o tempo de vida é marcado para *spill*.

  Se algum tempo de vida foi marcado para *spill* o alocador realiza o processo de *spill code* e reinicia a alocação. Se não é necessário *spill* ele avança para a etapa *select*.

- *Spill code* é invocado caso a etapa *simplify* decida realizar *spill* em alguma variável. São criados pequenos tempos de vida para cada uso da variável depois de sua definição, entre as instruções *load* e *store*.

- *Select* atribui uma cor para um vértice no grafo de acordo com a ordem estabelecida pela fase *simplify*. São repetidos os seguintes passo até que a pilha esteja vazia:

  - Tirar o tempo de vida do topo da pilha.
  - Inserir o tempo de vida no grafo.
  - Atribuir a este tempo de vida uma cor diferente da de seus vizinhos.

É importante observar que na remoção de um tempo de vida novos são gerados. Chaitin *et al.* [14] apontam que realizar *spill* em um tempo de vida não remove todas as suas interferências. Mesmo armazenando um valor na memória, ainda é necessário carregá-lo para um registrador antes de suas utilizações. Desta forma, é preciso garantir a disponibilidade de registradores entre operações *load* e *store*. Por causa disso, ao realizar *spill* de um tempo de vida, é necessário criar vários tempos de vidas menores para seus usos. Esses pequenos tempos de vida ainda podem interferir com os demais tempos de vida.

2.1.2 Alocador de Briggs

A coloração de grafos, e consequentemente também a alocação de registradores, é um problema NP-Completo, sendo necessário definir heurísticas para a sua resolução [13]. Neste contexto, várias pesquisas foram realizadas para melhorar os seus resultados. Entre estas pesquisas, Brigss *et al.* [1] descreveram a *optimistic coloring*, que consiste a utilização de heurísticas fortes para obter resultados k-colorável para o grafo de interferência. *Optimistic coloring* diminui o número de procedimentos que precisam de *spill* e reduz a quantidade de *spill* quando este é inevitável.

Optimistic coloring apresenta duas mudanças no alocador de Chaitin. Na etapa *simplify* é necessário que a remoção seja feita de acordo com grau dos vértices, de forma que o vértice com número de arestas maior ou igual ao número de cores esteja no topo da pilha. Além disso, tempos de vida marcados para *spill* são adicionados na pilha para uma possível coloração assim como os demais, em vez de realizar *spill* imediatamente. Por causa dessas mudanças, em *select* é possível que não exista cor disponível para um

vértice. Neste caso o vértice é deixado descolorido e marcado para *spill*, continuando com o próximo vértice. Em seguida é realizada a etapa *spill code*. É importante observar que, devido à ordem de prioridade, todos os vértices que seriam coloridos no alocador estilo Chaitin serão coloridos da mesma forma [1].

Desta forma a decisão de *spill* passa a ser realizada na etapa *select* em vez de *simplify*. Essa mudança é ilustrada na Figura 2.

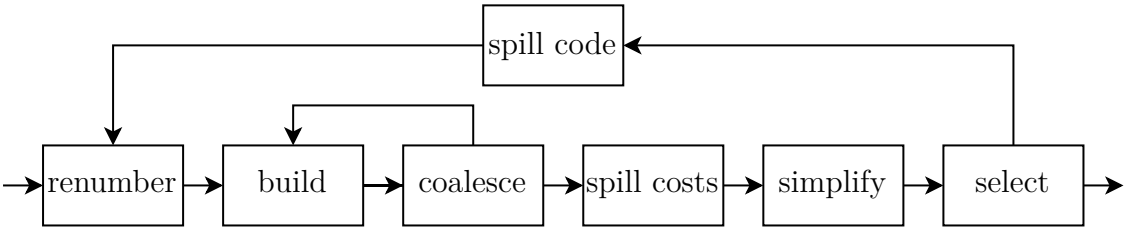


Figura 2 – Alocador estilo Briggs. Esta figura foi retirada do trabalho de Brigss *et al.* de [1].

Estas mudanças geram duas consequências. A primeira é a possibilidade tomar melhores decisões de *spill*, evitando *spills* improdutivos. A segunda é a possibilidade de colorir vértices com grau maior ou igual ao número de cores. É possível que 2 ou mais vizinhos possuam a mesma cor, desta forma estes vértices podem ter colorações válidas. Com o alocador de Chaitin eles seriam eliminados [1]. A Figura 3 apresenta um exemplo dessa situação, com vértices de grau três e coloração viável para três cores.

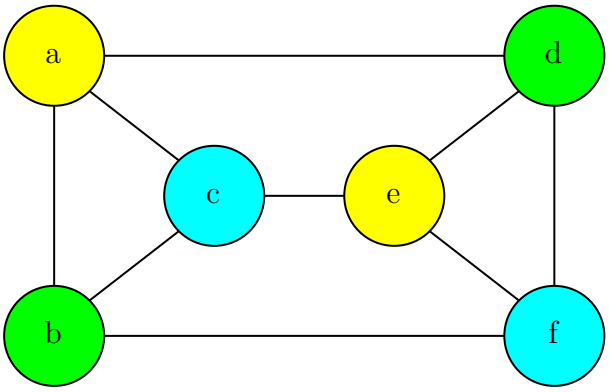


Figura 3 – Exemplo de grafo 3-colorável com vértices com 3 arestas.

2.1.3 Exemplo com o Alocador de Briggs

O exemplo de execução é feita considerando três cores (registradores) e o seguinte bloco simples de código:

```
a = 5;
b = a + 5;
c = 10 + a;
d = b + c;
```

```
e = b * d;
d = d + c;
print(e);
```

Inicialmente, é realizada a etapa *renumber*, obtendo os seguintes tempos de vida:

- A primeira definição de a está na primeira linha e o último uso na terceira;
- A primeira definição de b está na segunda linha e o último uso na quinta;
- A primeira definição de c está na terceira linha e o último uso na sexta;
- A primeira definição de d está na quarta linha e o último uso na sexta;
- A primeira definição de e está na quinta linha e o último uso na sétima;

Em seguida é necessário construir o grafo de interferência com a etapa *build*. O grafo de interferência está ilustrada na Figura 4. É importante considerar que é possível utilizar o mesmo registrador tanto para o cálculo quanto para a atribuição em uma operação. Por exemplo, a e c não interferem, apesar de serem usados na mesma instrução.

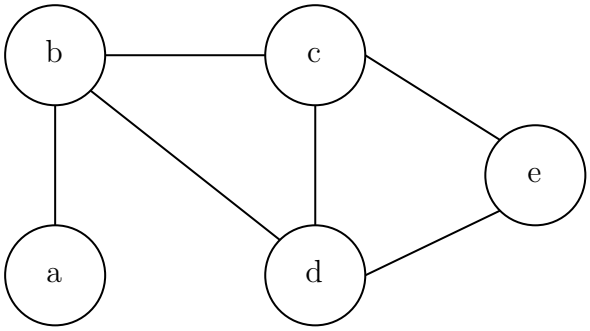
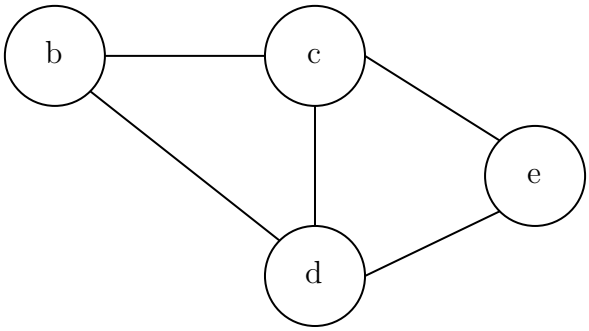


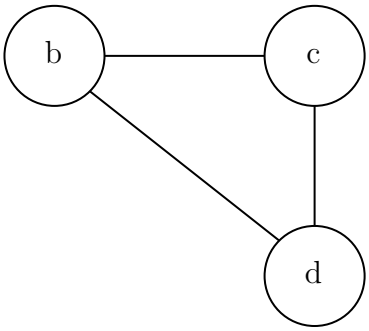
Figura 4 – Grafo de Interferência do bloco.

A etapa de *coalesce* não encontra nenhum caso que atenda as condições para combinar dois vértices. Assim, não ocorre nenhuma mudança no grafo de interferência, e o alocador segue para a fase *spill cost*. Essa etapa calcula o custo da realização *spill* para cada tempo de vida. Neste caso, c e d possuem o maior valor, com uma definição e duas utilizações. Os demais tempos possuem uma definição e uma utilização cada. Esses valores são utilizados caso seja necessário realizar decisões de *spill*.

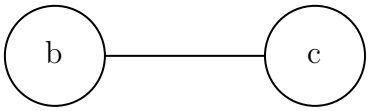
Em seguida, é feita a fase *simplify*. que remove os tempos de vida do grafo e os adiciona em uma pilha. A remoção é realizada sempre com o vértice com menor grau no grafo. Desta forma, a é removido inicialmente, seguido pelos demais em qualquer ordem. Essa etapa está ilustrado na Figura 5.



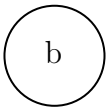
(a) Remoção de *a*.



(b) Remoção de *e*.



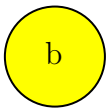
(c) Remoção de *d*.



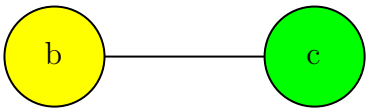
(d) Remoção de *c*, sobrando apenas *b*, que será removido em seguida.

Figura 5 – Remoção do vértices do grafo.

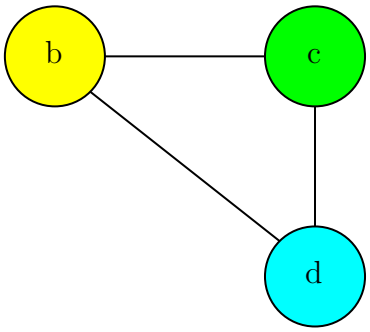
Após adicionar todos os tempos de vida na pilha, é realizada a etapa *select*. Nessa fase os vértices são retirados da pilha, coloridos e adicionados novamente no grafo. O resultado obtido é uma coloração válida com três registradores. Não é preciso realizar *spill* nesse exemplo. Esta etapa está ilustrada na Figura 6.



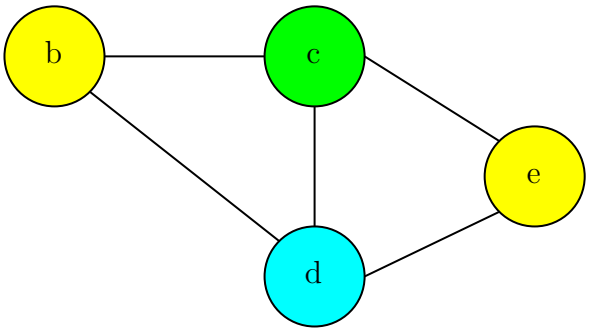
(a) Coloração e adição de *b*.



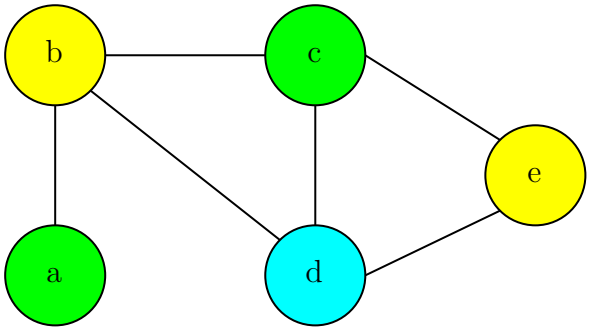
(b) Coloração e adição de *c*.



(c) Coloração e adição de *d*.



(d) Coloração e adição de *b*.



(e) Coloração e adição de *a*.

Figura 6 – Remoção dos vértices do grafo.

2.2 Linear Scan

Como uma alternativa para alocação de registradores baseada em coloração de grafo, foi proposto por Poletto e Sarkar [29] um algoritmo chamado *Linear Scan*. Esta abordagem é mais rápida, menos complexa e resulta em códigos quase tão eficientes quanto alocadores que usam coloração de grafo. Por causa disso se tornou uma opção interessante para aplicações na quais o tempo de compilação é uma preocupação, como compiladores dinâmicos e "*just-in-time*".

Linear Scan recebe o número de registradores k e uma lista de tempos de vida ordenados de acordo com o tempo inicial de forma ascendente. O número de registradores em interferência apenas muda quando algum tempo de vida termina ou começa. Desta forma, o algoritmo pode passar de um ponto de início de um tempo de vida para o próximo.

Em cada iteração é mantido uma lista, chamada *ativos*, de tempos de vida que interferem com o atual e foram atribuídos a um registrador. Essa lista é mantida ordenada de acordo com ponto finais dos tempos de vida de forma ascendente. Para cada novo ponto de início, *ativos* é percorrida para remover tempos de vida "expirados", que não interferem mais com o atual. Devido à ordenação, a verificação pode ser encerrada ao encontrar um tempo de vida que ainda interfira antes do fim da lista.

O tamanho dessa lista é, no máximo, o número de registradores. Caso isso acontece em uma nova iteração e nenhum tempo de vida seja removida de *ativos* é necessário realizar *spill*. Nessa situação é necessário escolher uma variável para ser armazenada na memória, entre o tempo de vida da iteração atual e as presentes em *ativos*. Existem várias heurísticas para realizar essa escolha. O Algoritmo 1 [29] apresenta o algoritmo de Linear Scan.

Algoritmo 1: Linear Scan

Entrada: Lista de tempos de vida $temposDeVida$, número de registradores k, lista de registradores.

Saída: Vetor associando tempo de vida e registrador alocado $registradores$.

```
1  início
2    para cada i ∈ temposDeVida faça
3      para cada j ∈ temposDeVida faça
4        se pontoFinal(j) ≤ pontoInicial(i) então
5          retorna;
6        fim
7        remove j de ativos;
8        adiciona registradores[j] na lista de registradores livres;
9      fim
10     se tamanho(ativos) = k então
11       Spill(i);
12       registradores[i] ← spill;
13     senão
14       registradores[i] ← um registrador removido da lista de
           registradores livres;
15       adiciona i para ativos, ordenado de forma ascendente pelo pontos
           finais;
16     fim
17   fim
18 fim
```

2.2.1 Exemplo de Linear Scan

Assim como na subseção 2.1.3, o exemplo de execução é feita considerando três registradores e o seguinte bloco simples de código:

```
a = 5;
b = a + 5;
c = 10 + a;
d = b + c;
e = b * d;
d = d + c;
print(e);
```

O algoritmo *linear scan* recebe a lista de tempos de vida ordenados de acordo com o tempo inicial de forma ascendente. Para isso, é necessário conhecer os pontos de início e de fim das variáveis. Os tempos de vida estão ilustrados na Figura 7:

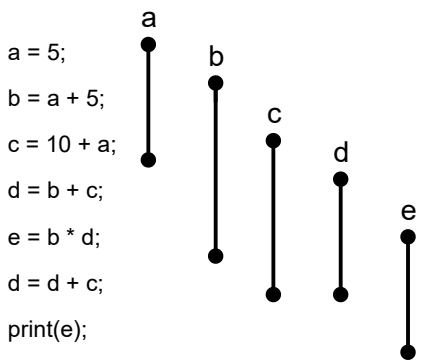


Figura 7 – Tempos de vida do exemplo de Linear Scan.

Assim, a lista de tempos de vida fica na forma:

$temposDeVida$ `= [a, b, c, d, e]`

No algoritmo essa lista é percorrida, realizando a alocação para cada tempo de vida. As iterações ocorrem na seguinte forma:

- Na primeira iteração a lista *ativos* está vazia, e a é associado a um registrador e adicionados em *ativos*;
- Na segunda iteração, não existem tempos de vida expirados em *ativos*, então *ativos* segue na forma:

  $ativos$ `= [a]`

  O tamanho de *ativos* é menor que três, então b é associado a um registrador e adicionados em *ativos*;
- Na terceira iteração a expira. Desta forma, é removida de *ativos*, que fica na forma:

  $ativos$ `= [b]`

  Como *ativos* continua com apenas um elemento, c é associado à um registrador e adicionados em *ativos*;
- Na quarta iteração nem b e nem c expiram, então *ativos* fica na forma:

  $ativos$ `= [b,c]`

  O tamanho de *ativos* continua menor que três, então d é associado a um registrador e adicionados em *ativos*;
- Por fim, na última iteração, ocorre a expiração de b, e *ativos* fica na forma:

  $ativos$ `= [c,d]`

Assim, o último tempo de vida e é associado a um registrador, e adicionado em *ativos*, pois *ativos* tem apenas dois elementos. Desta forma, é obtida uma coloração válida.

2.3 Partitioned Boolean quadratic programming (PBQP)

O PBQP (*Partitioned Boolean Quadratic Problem*) é um caso especial do *Quadratic Assignment Problem*. Consiste em um problema de múltipla escolha interdependentes e associadas a um custo. Existem algoritmos lineares (relativos ao número de arestas) que produzem resultados quase ótimos para grafos esparsos [2].

A representação mais utilizada do problema é por meio de grafos. Cada nó do grafo representa opções, sendo que exatamente uma deve ser escolhida. Cada nó está associado a um vetor com os custos de cada opção. As arestas representam as interdependências entre as escolhas, sendo associada a uma matriz de custo. Uma aresta entre um nó com vetor de dimensão n e outro com vetor de dimensão m terá uma matriz de dimensão $n \times m$ [2]. A Figura 8 apresenta um exemplo de coloração de grafos com PBQP.

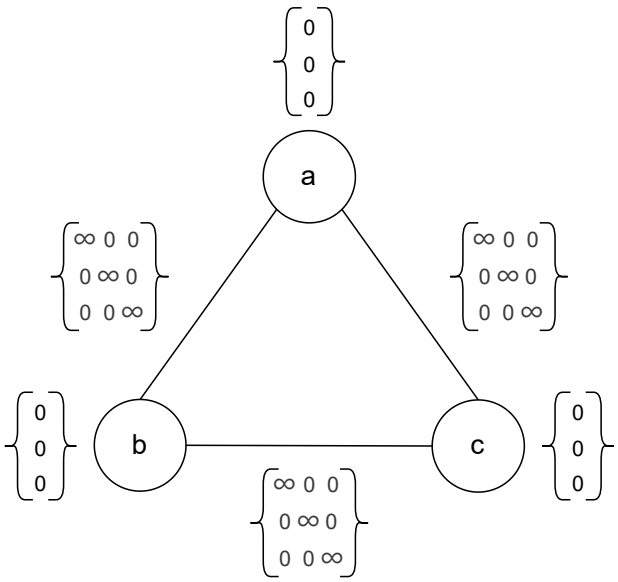


Figura 8 – Exemplo de coloração de grafos com PBQP. Essa figura foi retirada do trabalho de Buchwald *et al.* [2].

Selecionar opções em dois nós interdependentes significa selecionar implicitamente um valor de custo da matriz da aresta. Ao escolher a opção de índice i de um nó e opção j de outro significa selecionar o custo de da linha i e coluna j da matriz da aresta. No exemplo da Figura 8 as diagonais das matrizes possuem valores infinitos, pois nós vizinhos não podem ter a mesma cor. Se a soma de todos os custos selecionados é finito, a

seleção realizada é considerada uma solução. O objetivo do PBQP é encontrar a solução de mínimo custo [2].

Considerando as cores como registradores e os nós os tempos de vida das variáveis, é possível abstrair a alocação de registradores para PBQP. A principal vantagem dessa abordagem é a sua flexibilidade devido ao custo associado as decisões de coloração, se tornando vantajosa para arquiteturas irregulares [9].

Essa abordagem foi implementada primeiramente por Scholz e Eckstein [30], contendo uma heurística linear. Foi proposta uma solução com programação dinâmica dividida em três fases. Na primeira o PBQP é reduzido em subproblemas. Em cada redução, vetores de decisões são eliminados até que a solução seja trivial. Na segunda fase são definidas as escolhas de custo e a solução. Por fim, na terceira fase ocorre a propagação da solução da fase anterior para realizar decisões de custo nos vetores eliminados. As possíveis formas de redução são [30, 31, 32]:

- **RE:** Arestas independentes podem ser removidas após a sua matriz de custo ser decomposta em dois vetores e esses serem somados aos vetores de custo dos vértices. Geralmente não é possível usar essa remoção considerando o problema de alocação de registradores, resultando na remoção de uma interferência;

Figura 9 – Exemplo de remoção RE no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2].

- **R1:** Vértices de grau 1 podem ser removidos após terem os custos considerados no vértice adjacente;

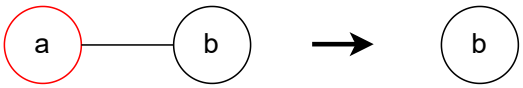


Figura 10 – Exemplo de remoção R1 no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2].

- **R2:** Vértices de grau 2 podem ser removidos após terem os custos considerados na matriz de custo na aresta entre os vértices adjacentes criada com a remoção;

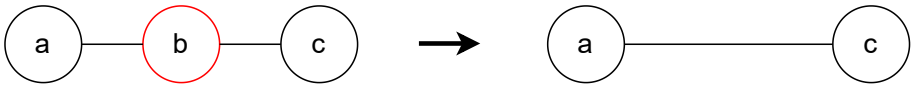


Figura 11 – Exemplo de remoção R2 no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2].

- **RN:** Para vértices com grau maior que 2 é necessário aplicar uma heurística para realizar a redução.

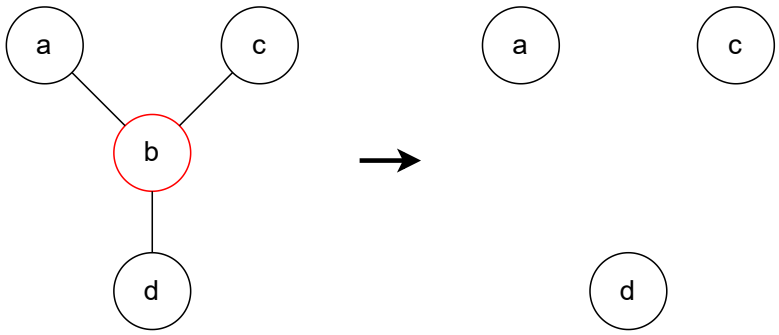


Figura 12 – Exemplo de remoção RN no PBQP. Esta figura foi retirada do trabalho de Buchwald *et al.* [2].

As reduções são aplicadas até que o grafo não tenha interferência, tornando a solução trivial. RE, R1 e R2 são reduções ótimas, eliminando vértices do grafo PBQP, mas mantendo o custo mínimo. Desta forma a solução de instâncias menores podem ser estendidas para solução da instância original com custo igual. Para grafos esparsos essas reduções são bastante efetivas. Caso o grafo inteiro seja redutível por RE, R1 e R2, então o resultado do PBQP é ótimo. Se algum nó de grau 3 ou maior permanecer é necessário usar a heurística de RN. Neste caso a solução ótima é perdida, mas garante tempo linear para grafos esparsos.

Hames e Scholz [33] apresentaram uma nova heurística RN para melhorar a qualidade do código gerado. Os vértices candidatos são avaliados e colocados em uma pilha, sendo que a decisão de remoção ocorre apenas na fase de propagação.

Além disso, introduziram uma técnica *branch-and-bound* para PBQP para soluções ótimas. *Branch-and-bound* é uma técnica para resolver problemas de otimizações combinatórias discretas [33]. *Branch-and-bound* consiste em dois conceitos. O primeiro é *branching*, a decomposição de um problema em sub-problemas. *Branching* é aplicado recursivamente para cada sub-problema, formando uma *search tree*. O segundo é *Bounding*, uma forma rápida de achar ligações entre os níveis da árvore e eliminar caminhos não ótimos, reduzindo o número de sub-problemas explorados. No contexto do PBQP, *Branching* consiste em reduções no grafo e *Bounding* em heurísticas para encontrar as melhores reduções e filtrar as soluções analisadas.

2.3.1 Exemplo de PBQP

Assim como na subseção 2.1.3, o exemplo de execução é feita considerando três registradores e o seguinte bloco simples de código:

```
a = 5;
b = a + 5;
c = 10 + a;
d = b + c;
e = b * d;
d = d + c;
print(e);
```

Similar a grafo no exemplo do alocador de Briggs da Figura 4, o grafo PBQP desse código está ilustrado na Figura 13.

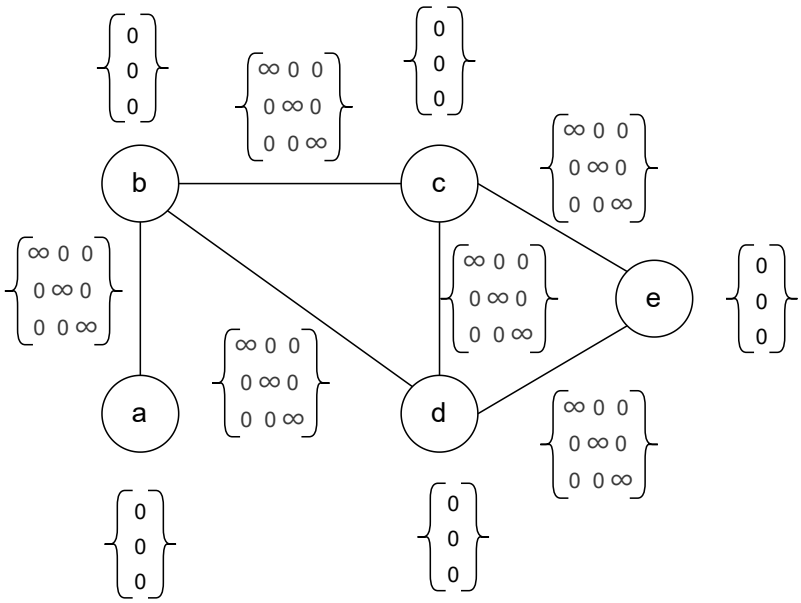


Figura 13 – Grafo PBQP do exemplo.

Para resolver a alocação de registradores por PBQP, é necessário reduzir o grafo até achar uma solução trivial. Isso, é feito através de simulações e propagações de diferentes decisões de remoções. Assim, considerando que aé removido inicialmente utilizando R1. O grafo resultante é apresentado na Figura 14.

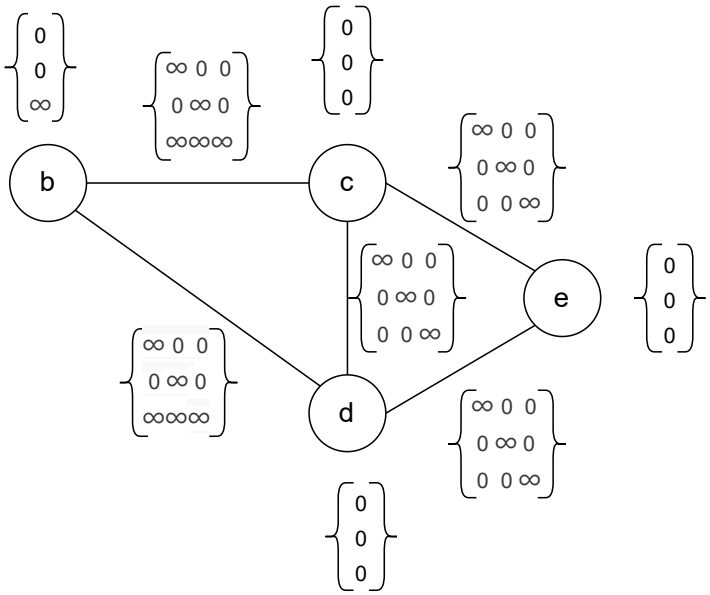


Figura 14 – Grafo PBQP do exemplo após a remoção de a.

Em seguida, b é removido utilizando R2. O grafo resultante é apresentado na Figura 15.

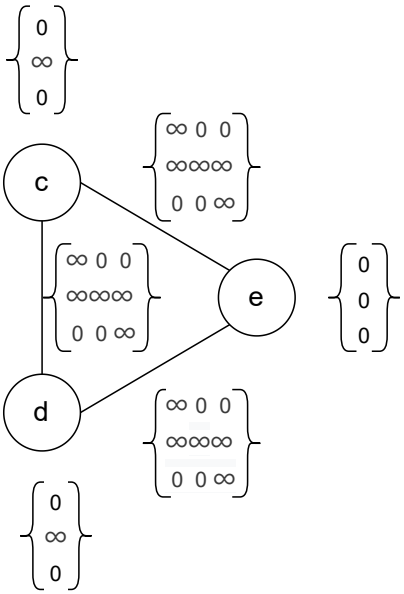


Figura 15 – Grafo PBQP do exemplo após a remoção de b.

A próxima remoção é realizada em c utilizando R2. O grafo resultante é apresentado na Figura 16.

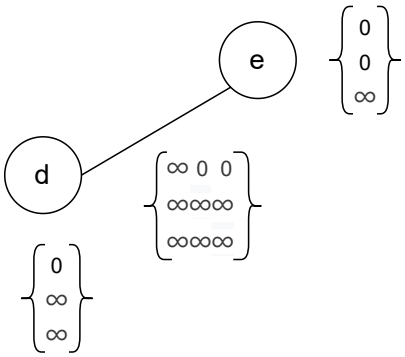


Figura 16 – Grafo PBQP do exemplo após a remoção de *c*.

A última remoção é realizada em *c* utilizando R2 novamente. O grafo resultante é apresentado na Figura 17.

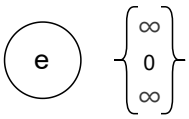


Figura 17 – Grafo PBQP do exemplo após a remoção de *d*.

Desta forma, é encontrada uma solução ótima com custo zero. Como estas remoções já conseguiram o melhor resultado possível, não é necessário explorar outras possibilidades de reduções.

2.4 Comparação entre as Abordagens de Alocação Global

Nas seções anteriores foram apresentadas as três abordagens de alocação de registradores globais mais utilizadas [11]. Nesta seções essas técnicas serão comparadas, apresentando suas complexidades, desempenho e principais utilizações.

O desempenho é apresentado nas Tabelas 1 e 2, utilizando o número de código de *spill* inserido, ou seja, a quantidade de instruções *load* e *store* adicionadas. A tabela foi retirada da dissertação de mestrado de Lopes da Silva [10], no qual foram compilados programas do *benchmark* SPEC CPU2006 para arquitetura x86_64 e ARM Cortex-A8. Os testes são realizados utilizando o *framework* do LLVM [34]. São avaliados os seguintes alocadores:

- **Briggs**: Alocador de registradores por coloração de grafos utilizando *optmistic coloring* de Briggs *et al.* [1] implementado por Lopes da Silva [10];

- **Basic**: Opção de *Linear Scan* presente no LLVM [9];
- **Greedy**: Opção de *Linear Scan* com *live range splitting* (Seção 3.5) agressivo presente no LLVM [9];
- **PBQP**: Alocador PBQP presente no LLVM [9].

Tabela 1 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura x86_64, usando os alocadores Chaitin-Briggs e os alocadores do LLVM. Esta tabela foi retirada da dissertação de mestrado de Lopes da Silva [10].

SPEC CINT2006				
Benchmark	**Briggs**	**Greedy**	**Basic**	**Pbqp**
`400.perlbench`	2.957	3.789	3.568	3.192
`401.bzip2`	323	531	329	309
`403.gcc`	6.422	7.352	7.527	7.396
`429.mcf`	21	17	20	22
`445.gobmk`	2.230	2.365	2.325	2.230
`456.hmmer`	1.389	1.205	1.424	1.388
`458.sjeng`	196	236	217	196
`464.h264ref`	2.908	3.068	3.014	2.867
`471.omnetpp`	737	583	759	724
`473.astar`	190	176	197	189
Total	17.373	19.322	19.380	18.513
SPEC CFP2006				
Benchmark	**Briggs**	**Greedy**	**Basic**	**Pbqp**
`433.milc`	663	612	693	677
`444.namd`	4.813	5.055	5.087	4.731
`450.soplex`	1.255	1.127	1.310	1.261
`470.lbm`	89	41	89	91
Total	6.820	6.835	7.179	6.760

Tabela 2 – Quantidade de código *spill* inserida para cada *benchmark* do SPEC CPU 2006 para a arquitetura ARM Cortex-A8, usando os alocadores Chaitin-Briggs e os alocadores do LLVM. Esta tabela foi retirada da dissertação de mestrado de Lopes da Silva [10].

SPEC CINT2006				
Benchmark	**Briggs**	**Greedy**	**Basic**	**Pbqp**
`400.perlbench`	2.684	3.337	3.271	3.260
`401.bzip2`	571	739	573	539
`403.gcc`	6.661	7.589	7.605	7.694
`429.mcf`	30	31	36	31
`445.gobmk`	2.000	2.311	2.216	2.148
`456.hmmer`	723	855	755	783
`458.sjeng`	415	492	464	422
`464.h264ref`	3.799	3.984	3.981	3.818
`471.omnetpp`	192	239	196	236
`473.astar`	230	216	236	226
Total	17.305	19.793	19.333	19.157
SPEC CFP2006				
Benchmark	**Briggs**	**Greedy**	**Basic**	**Pbqp**
`433.milc`	466	491	462	486
`444.namd`	3.655	4.926	3.759	3.569
`450.soplex`	772	902	840	829
`470.lbm`	28	22	32	28
Total	4.921	6.341	5.093	4.912

A coloração de grafos é o método mais utilizado para alocação de registradores [11]. Como foi a primeira técnica desenvolvida, diversas pesquisas foram realizadas para melhorar a coloração. Conforme observado nas Tabelas 1 e 2, a abordagem de Briggs apresentou melhores resultados entre os alocadores. No SPEC CINT 2006 obteve o menor número total de código de *spill* inserido para as duas arquiteturas, com um total 10% menor que as outras abordagens. Ainda, no SPEC CFP 2006 apresentou resultados muito próximos aos do melhor alocador. Por outro lado, a abordagem de Briggs é uma técnica custosa para o compilador, com complexidade $O(n^2)$, sendo n o número de vértices [28].

O *Linear Scan* é um algoritmo mais rápido, com complexidade $O(nlog\ r)$, sendo n o número de vértices e r o número de registradores [29]. Sua velocidade é interessante para aplicações na quais o tempo de compilação é uma preocupação, como compiladores dinâmicos e "*just-in-time*". Por outro lado, isso resulta em um desempenho um pouco pior que as outras técnicas, o que pode ser observado pelo número elevado de inserções de código de *spill* comparado às outras abordagens.

Por fim, o PBQP apresenta resultados próximos das outras técnicas. Em relação ao alocador de Briggs o alocador PBQP obteve resultador piores. Apresentou números totais de código de *spill* inseridos melhores no SPEC CFP 2006 por uma margem baixa. Porém,

obteve resultados considerávelmente piores no SPEC CINT 2006. Sua complexidade, considerando os algoritmos mais utilizados para o problema de alocação de registradores, é $O(nr^3 + mr^2)$, sendo n o número de vértices, m o número de arestas e r o número de registradores [11, 2]. A Alocação de registradores por PBQP é vantajosa para arquiteturas irregulares devido a sua flexibilidade [9].

2.5 Alocação de Registradores Local

As abordagens apresentadas nas Seções 2.1, 2.2 e 2.3 são consideradas técnicas globais. Alternativamente, existem métodos para realizar alocação de registradores local, que consideram apenas um bloco básico de código, ou seja, uma sequência de instruções sem nenhum tipo de desvio.

O principal objetivo da alocação de registradores local é minimizar o tráfico de dados entre o CPU e a memória, buscando a melhor distribuição possível de operações *load* e *store* no código [35, 36]. Apesar de considerar apenas um bloco do código, é possível estender a alocação local para global [37, 38, 39].

Antes do processo de alocação, todos os valores possíveis no código são salvos em registradores, utilizando quantos nomes de registradores forem necessários. Sendo assim, é possível que sejam nomeados mais registradores do que a quantidade presente no processador. Por causa disso, os registradores nomeados nessa fase são chamados de registradores virtuais [27].

Desta forma, a alocação de registradores local precisa criar um bloco equivalente onde as referências para registradores virtuais sejam substituídas para referências a registradores reais. Se o número de registradores virtuais for maior que o número de registradores reais é necessário realizar *spill* por meio dos comandos *load* e *store*.

Uma possível abordagem é a alocação de registradores local *bottom-up*, que foca nas transições que ocorrem na execução de cada operação. Para cada operação, o alocador deve garantir que os operadores estejam em registradores antes da execução, além de alocar o resultado em outro. Para isso, o alocador percorre todas as operações no bloco verificando a necessidade de *spill* [27].

O algoritmo começa com todos os registradores livres, armazenados em uma lista. As operações do bloco são percorridas, e os registradores virtuais utilizados são substituídos por registrados disponíveis, que são removidos da lista. O algoritmo mantêm salva uma referência para a próxima utilização do registrador virtual. Isso permite que em sua última utilização o registrador físico associado retorne para a lista de registradores livre. Se a lista estiver vazia, é escolhido um valor para sofrer *spill*. A decisão é feita com base na posição do próximo uso do valor, sendo que o valor com próximo uso mais distante é escolhido. Desta forma, o registrador fica livre pelo maior tempo possível antes que o

valor seja recarregado da memória [27].

2.5.1 Exemplo de Alocação de Registradores Local

O exemplo de execução é realizado considerando três registradores e o seguinte código:

```
a = 5;
b = 10;
c = 15;
d = a + b;
a = b + d;
b = 2 * a;
d = d + c;
```

Após a substituição das variáveis pelos registradores virtuais o código fica na forma:

```
v1 = 5;
v2 = 10;
v3 = 15;
v4 = v1 + v2;
v1 = v2 + v4;
v2 = 2 * v1;
v4 = v4 + v3;
```

Durantes as três primeiras instruções, apenas são substituídos os registradores virtuais por físicos, resultando no seguinte código:

```
r1 = 5;
r2 = 10;
r3 = 15;
v4 = v1 + v2;
v1 = v2 + v4;
v2 = 2 * v1;
v4 = v4 + v3;
```

Na quarta iteração, é necessário realizar *spill*, pois não existem registradores físicos disponíveis para *d*. Desta forma, *c* é enviado para a memória, pois seu próximo uso é o mais distante. Após este processo, o código fica na seguinte forma:

```
r1 = 5;
r2 = 10;
r3 = 15;
store r3;
```

```
v4 = v1 + v2;
v1 = v2 + v4;
v2 = 2 * v1;
v4 = v4 + v3;
```

Nas próximas duas instruções os registradores físicos alocados são mantidos, conforme apresentado abaixo:

```
r1 = 5;
r2 = 10;
r3 = 15;
store r3;
r3 = r1 + e2;
r1 = r2 + r3;
v2 = 2 * v1;
v4 = v4 + v3;
```

Na seguinte instrução os valores necessários já estão armazenados em registradores físicos. Além disso, dois registradores físicos são disponibilizados, devido ao fim dos tempos de vida de a e a. O código após esta etapa fica na forma:

```
r1 = 5;
r2 = 10;
r3 = 15;
store r3;
r3 = r1 + e2;
r1 = r2 + r3;
r2 = 2 * r1;
v4 = v4 + v3;
```

Por fim, o valor c é recarregado da memória para o registrador *r1*. Em seguidas, os registradores virtuais são substituídos por registradores físicos na última instrução. Assim, o código resultante é:

```
r1 = 5;
r2 = 10;
r3 = 15;
store r3;
r3 = r1 + r2;
r1 = r2 + r3;
r2 = 2 * r1;
load r1;
r3 = r3 + r1;
```

2.5.2 Diferenças entre Alocação Local e Global

Comparando com abordagens globais, a alocação local apresenta maior precisão e eficiência para alocar registradores em blocos simples de código, possibilitando lidar diretamente com operações de armazenamento e recuperação na memória em vez de tempos de vida [36].

Por outro lado, abordagem globais são mais adequadas para alocar registradores em múltiplos blocos, pois consideram o valor de reutilização de variáveis através de vários blocos. A alocação de registradores local enfrenta dificuldades para lidar com o fluxo de valores entre blocos, muitas vezes resultando em alocações ineficientes [27].

3 MINIMIZAÇÃO DE SPILL

O principal objetivo da alocação de registradores é evitar o armazenamento de variáveis na memória, sendo consideravelmente menos custoso utilizar registradores. Porém, mesmo utilizando os alocadores mais eficientes possíveis, ainda é necessário enviar alguns valores para a memória na maioria dos casos. Desta forma, é necessário minimizar o máximo possível o custo da realização de *spill*, reduzindo a inserção de instruções *store* e *load* no código [40, 41].

Uma forma de diminuir o custo de *spill* é por meio da escolha do tempo de vida que será enviado para a memória. Tempos de vida apresentam diferentes números de usos e definições, afetando diretamente a quantidade de código de *spill* necessário. Ademais, a remoção de diferentes vértices resultam em grafos de interferência diferentes. Boas heurísticas de remoções podem evitar a necessidade de realizar *spill* novamente.

Além disso, é possível desenvolver diversos mecanismos para reduzir a adição de instruções *store* e *load*. A escolha de boas posições, por exemplo, de instruções *store* e *load* pode evitar inserções excessivas. Ainda, a divisão de tempos de vida pode evitar interferência, permitindo melhor utilização de registradores.

Esse e outros métodos de minimização de *spill* podem ter um impacto significativo no desempenho do código gerado pelo compilador. Sendo assim, além de buscar abordagens para melhorar o mapeamento de variáveis, é interessante pesquisar métodos para diminuir o custo de *spill* quando este é inevitável. As próximas seções deste capítulo apresentam as principais técnicas descritas na literatura.

3.1 Heurísticas de Chaitin

Inicialmente, o alocador de Chaitin [14] tinha heurísticas pouco eficientes para realizar *spill*. O alocador insere uma quantidade excessiva de código de *spill*, adicionando uma instrução *store* por definição e uma instrução *load* no começo de cada bloco simples seguinte no tempo de vida. Posteriormente, Chaitin *et al.* [42] apresentou novas heurísticas para melhorar seu alocador. Essas mudanças resultaram em menor inserção de código *spill* e redução no tempo de compilação.

Para possibilitar melhores decisões de *spill*, foi proposta uma heurística para estimar o custo de armazenar certo tempo de vida na memória. O cálculo é baseado na suposição que instruções em *loops* custam dez vezes mais que instruções fora. Assim, o peso de cada instrução de definição ou de uso é a sua profundidade em *loops*. Uma utilização com profundidade dois, ou seja, dentro de dois *loops*, apresenta peso cem.

Além disso, o custo é divido pelo grau do vértice, resultando no valor da heurística h, que é avaliado para decisões de *spill*. Isso permite escolher tempos de vida que inserem menos código de *spill* e eliminam o maior número de interferências no grafo. Desta forma, o cálculo dessa heurística é descrito nas seguintes equações:

$$custo(v) = \sum_{I \in \text{definições e usos de } v} 10^{profundidade(I)}$$

$$h(v) = \frac{custo(v)}{grau(v)}$$

Ainda, Chaitin *et al.* [42] apresentou o conceito de proximidade entre instruções que usam um tempo de vida que sofreu *spill*. Duas utilizações estão próximas se nenhum registrador ficar disponível entre elas. Novos registradores apenas ficam disponíveis após o fim de algum tempo de vida. Assim, duas instruções são próximas se não ocorrer o fim de outro tempo de vida entre elas.

É importante observar que se outro tempo de vida for iniciado entre duas instruções próximas, ele utilizaria outro registrador. Como o novo tempo de vida não acaba antes da segunda utilização, ocorre interferência, garantindo diferente coloração entre eles. Além disso, como não foram disponibilizados novos registradores entre as instruções, o registrador da segunda instrução está disponível para a primeira. Sendo assim, é possível alocar o mesmo registrador para as duas utilizações, de forma que o valor é mantido entre elas. Assim, é desnecessário adicionar a instrução *load* na segunda utilização [3].

Desta forma, foram introduzidas diretrizes locais para reduzir o número de instruções *store* e *load* adicionadas na realização de *spill*:

- Se uma definição e um uso estão próximos, é desnecessário adicionar uma instrução *load* antes do uso;
- Se dois usos estão próximos é adicionado uma instrução *load* antes do primeiro uso apenas;
- Se a primeira definição e o último uso de um tempo de vida estão próximos, sua remoção não ajuda na coloração do grafo. Isso ocorre pois nenhum registrador foi liberado, mantendo o número de interferências anterior. Sendo assim, o tempo de vida não deve ser enviado para a memória, e seu custo de *spill* é infinito.

Além disso, Chaitin *et al.* [42] introduziu uma versão básica de *rematerialization*, que será discutida na Seção 3.3.

3.2 Heurísticas de Bernstein

Bernstein *et al.* [40] apresentaram novos métodos heurísticos que minimizam a quantidade de código de *spill* gerado pelo alocador. Foram propostas novas fórmulas para decisões de *spill* (*best-of-three*), uma técnica para diminuir a inserção de instruções *store* e *load* em certas regiões (*cleaning*) e um novo método para definir a ordem de coloração de vértices (estratégia gulosa de coloração).

Best-of-three: Dependendo das características dos tempos de vida, enviá-los para a memória poder afetar a disponibilidade de registradores de diversas formas. *Spill* de tempos de vida com baixo custo resultam em menor perda de desempenhos, enquanto eliminação de tempos de vida com alto grau permitem remoção de mais interferências, diminuindo a possibilidade de novos *spills*. Desta forma, é difícil achar uma heurística que consiga balancear as diferentes características.

Neste contexto, Bernstein *et al.* [40] propuseram o cálculo de três heurísticas diferentes para decisões de *spill*. Para cada tempo de vida, são calculados os três valores e o melhor entre eles é considerado para avaliar o possível *spill*.

A primeira heurística é similar à heurística de Chaitin *et al.* [42], mas com ênfase em enviar para memória tempos de vida com grau elevado para minimizar as interferências dos grafos. O cálculo da heurística é descrita pela seguinte equação:

$$h_1(v) = \frac{custo(v)}{grau(v)^2}$$

A segunda e terceira heurísticas buscam por tempos de vida que ficam vivos por mais tempo, diminuindo a pressão de registradores. Para isso foi desenvolvido o cálculo da *área* do tempo de vida, dada pela seguinte equação:

$$area(v) = \sum_{\substack{I \in \text{instruções} \\ v \text{ está viva em I}}} 5^{profundidade(I)} largura(I)$$

O valor *largura* é o número de variáveis vivas durante a execução da instrução. As fórmulas para estas heurísticas são as seguintes:

$$h_2(v) = \frac{custo(v)}{area(v)grau(v)}$$

$$h_3(v) = \frac{custo(v)}{area(v)grau(v)^2}$$

Cleaning: Bernstein *et al.* [40] apresentaram uma abordagem para minimizar a inserção de código de *spill*. Para cada bloco básico, apenas uma instrução *load/store* é

introduzida para o primeiro uso/definição do tempo de vida que sofreu *spill*. Em seguida, o registrador utilizado é renomeado, gerando um pequeno tempo de vida limitado no bloco básico.

Estratégia gulosa de coloração: Bernstein *et al.* [40] apontam que a coloração pode ser feita de forma gulosa, e que uma ordem gulosa pode permitir melhores colorações. Diferentes regiões de um código formam subgrafos, que podem ser coloridos com base em heurísticas locais e combinados para formar uma solução heurística. Desta forma, Bernstein *et al.* propuseram que pequenos procedimentos chamados frequentemente devem ser coloridos com o menor número de registradores possíveis, aumentando a disponibilidade de registradores para outras regiões do código. Ainda, foi proposto maior expansão *inline* [43] para procedimentos pequenos com baixa frequência de uso.

Comparadas ao compilador de Chaitin, as heurísticas de Bernstein *et al.* reduziram em média de 6% a 12% o número de instruções de *spill* inseridos, chegando a 30% em alguns casos.

3.3 Rematerialization

Quando um valor é marcado para *spill* o alocador deve verificar se é menos custoso calcular novamente o valor do que armazená-lo na memória para ser recuperado posteriormente. Esse problema é chamado de *Rematerialization* [44]. Chaitin *et al.* [14] apontaram que certos valores podem ser recalculados em uma única instrução e que os operandos necessários sempre estão disponíveis. Como recalcular é menos custoso que armazenar e recarregar os valores da memória, eles não devem sofrer *spill*. Ainda, observaram que cópias desses valores que não sofreram *coalesce* podem ser eliminadas através do cálculo do valor direto no registrador desejado. Na prática, oportunidades para *rematerialization* incluem [1]:

- *Load* imediato de constantes inteiras e, dependendo da máquina, de constantes ponto flutuante.
- Cálculo de desvios constantes em ponteiros.
- *Load* em um endereço constante.

É importante observar a diferença entre valores e tempos de vida de variáveis. Um tempo de vida contém vários valores durante seu uso. Para realizar *rematerialization* é preciso verificar que é possível calcular novamente o valor. O alocador de Chaitin realiza o processo corretamente em tempos de vida com um único valor. mas não em casos mais complexos com múltiplos valores.

Neste contexto, Brigss *et al.* [1] propuseram uma abordagem para lidar com esses casos. Seu método consiste em dividir os tempos de vida pelos seus valores, rotular cada valor com informações para o processo de *rematerialization* e formar novos tempos de vida com valores com marcações idênticas.

Para achar cada valor é construído o grafo *static single assignment* (SSA) [45] do procedimento. Essa representação resulta em um código no qual cada variável possui exatamente uma atribuição (definição). Desta forma cada variável está associada a um único valor. Para criar essa representação é preciso criar definições chamadas ϕ-nodes em pontos do código onde ocorrem a combinação de diferentes valores. Desta forma, cada definição de valor ou é uma instrução simples ou um ϕ-node que combina dois ou mais valores.

Verificando as instruções de definição de valores é possível identificar candidatos para *rematerialization* e rotulá-los. É utilizado um algoritmo similar ao *sparse simple constant algorithm* (SSPC) [46] para propagar os rótulos através do grafo. Os possíveis rótulos são:

- $\top$: Significa que não são conhecidas informações do vértice. Valores definidos por instruções de cópias e ϕ-node são inicializados com este rótulo;
- *inst*: Significa que a instrução da definição é apropriada e o valor deve sofrer *rematerialization*. O valor do rótulo é um ponteiro para a instrução;
- $\bot$: Significa que o valor deve sofrer *spill*.

É necessário criar regra para o encontro entre rótulo durante a varredura do grafo. As regras de encontro são:

- Caso um rótulo qualquer encontre outro rótulo, $\top$ a marcação do primeiro é mantido;
- Caso um rótulo qualquer encontre outro rótulo, $\bot$ a marcação resultante é $\bot$;
- Caso dois rótulos *inst* com valores iguais se encontrem o rótulo e o ponteiro são mantidos;
- Caso dois rótulos *inst* com valores diferentes se encontrem o rótulo resultante é $\bot$.

Após a propagação, rótulos $\top$ serão substituídos por *inst* ou $\bot$. Valores definidos por instruções de cópias apresentam o mesmo rótulo que o original. ϕ-nodes têm rótulos *inst* se os valores que são combinados tem marcação *inst* e apontam para a mesma definição. Caso contrário, apresentam rótulos $\bot$.

Em seguida, é realizada a remoção de ϕ-nodes para recriar um código executável. Esse processo é realizado através da criação de uma cópia da variável. Devido ao processo anterior, é possível que sobrem cópias desnecessárias de variáveis. Por causa disso, valores com mesma marcação são identificados e combinados.

Por meio das informações marcadas o alocador consegue realizar corretamente o processo de *rematerialization* em tempos de vida com múltiplos valores. Entretanto, essa abordagem gera outro problema, realização de divisões improdutivas. As divisões podem criar valores redundantes, que é adicionado apenas para ser atribuído para outro valor. Assim, são adicionadas instruções de cópias desnecessárias no código. Para minimizar esse problema, são adotadas duas medidas no alocador. O processo de *coalesce* apenas acontece caso o tempo de vida resultante não sofra *spill*. Além disso, a etapa *select* tenta atribuir a mesma cor para tempos de vida conectados por uma cópia [1].

Briggs *et al.* [1] executaram 70 *benchmarks*, demonstrando que a rematerialização permite redução considerável de *spill*. Em 28 casos houve melhorias que ultrapassaram 20%, e apenas dois casos inserindo mais código de *spill* que a heurística de Chaitin.

3.3.1 Exemplo de Rematerialization

O exemplo de *rematerialization* é feito considerando o seguinte código:

```
a = 10;
i = 0;
while(i < 10){
    b = b + a;
    i = i + 1;
}
j = 0;
while(j < 10){
    a = a + 1;
    j = j + 1;
}
```

Inicialmente é realizada a transformação para SSA, resultando no seguinte código:

```
a1 = 10;
i1 = 0;
while(φ(i1,i2) < 10){
    b = b + a1;
    i2 = φ(i1,i2) + 1;
}
j1 = 0;
```

```
while(φ(j1,j2) < 10){
    a2 = φ(a1,a3) + 1;
    j2 = φ(j1,j2) + 1;
}
```

Em seguida ocorre a varredura e rotulação dos valores. Os rótulos dos valores são:

- $a1$: $inst(a1 = 10)$;
- $i1$: $inst(i1 = 0)$;
- $j1$: $inst(j1 = 0)$;
- $b2$: $\perp$;
- $a2$: $\perp$;
- $i2$: $\perp$;
- $j2$: $\perp$;
- $\phi(a1, a2)$: $\perp$;
- $\phi(i1, i2)$: $\perp$;
- $\phi(j1, j2)$: $\perp$;

Após essa etapa, os ϕ-nodes são removidos:

```
a1 = 10;
i1 = 0;
i2 = i1;
while(i2 < 10){
    b = b + a1;
    i3 = i2 + 1;
    i2 = i3;
}
j1 = 0;
a2 = a1;
j2 = j1;
while(j2 < 10){
    a3 = a2 + 1;
    a2 = a3;
    j3 = j2 + 1;
    j2 = j3;
}
```

Por fim, são removidas as cópias desnecessárias, resultando no seguinte código:

```
a1 = 10;
i1 = 0;
i2 = i1;
while(i2 < 10){
    b = b + a1;
    i2 = i2 + 1;
}
j1 = 0;
a2 = a1;
j2 = j1
while(j2 < 10){
    a2 = a2 + 1;
    j2 = j2 + 1;
}
```

Assim, é possível continuar o processo de alocação de registradores. Os rótulos são utilizados posteriormente para calcular o custo de *spill*. Além disso, são utilizados na inserção de código *spill*, possibilitando adição de instruções para recalcular o valor em vez de armazená-lo na memória. Considerando este exemplo, $a1$, $i1$ e $j1$ são valores que poderiam sofrer *rematerialization*. No entanto, $i1$ e $j1$ são divisões improdutivas geradas pelo processo. Esses valores deveriam ser combinados com $i2$ e $j2$, formando tempos de vida únicos, similares a i e j presentes no código original. Não existe benefício em realizar *rematerialization* em $i1$ e $j1$. Sendo assim, apenas $a1$ deveria sofrer *rematerialization*.

3.4 Spilling por Região de Interferência

Bergner *et al.* [3] apresentaram a eliminação da região de interferência, uma nova abordagem para minimizar o custo de *spill*. Considerando a interferência entre dois pontos de vida, a região de interferência consiste na parte do código onde ambas estão vivas simultaneamente. Eliminando essa região de um dos tempos de vida através de *spill* elas deixam de se interferir, permitindo que os dois tempos de vida possam usar o mesmo registrador. Assim, a alocação de registradores é facilitada, evitando o *spill* completo de uma das variáveis.

A eliminação da região de interferência é realizada através do *spill* parcial de um dos tempos de vida. A inserção do código de *spill* é limitada, considerando apenas os usos do tempo de vida dentro da região, além das definições da variável que podem alcançar a região de interferência [3]. Desta forma, o número de instruções *store* e *load* é reduzida para tempos de vida que não podem sofrer *rematerialization* (Seção 3.3). A Figura 18

apresenta a diferença entre *spill* total do tempo de vida e *spill* da região de interferência.

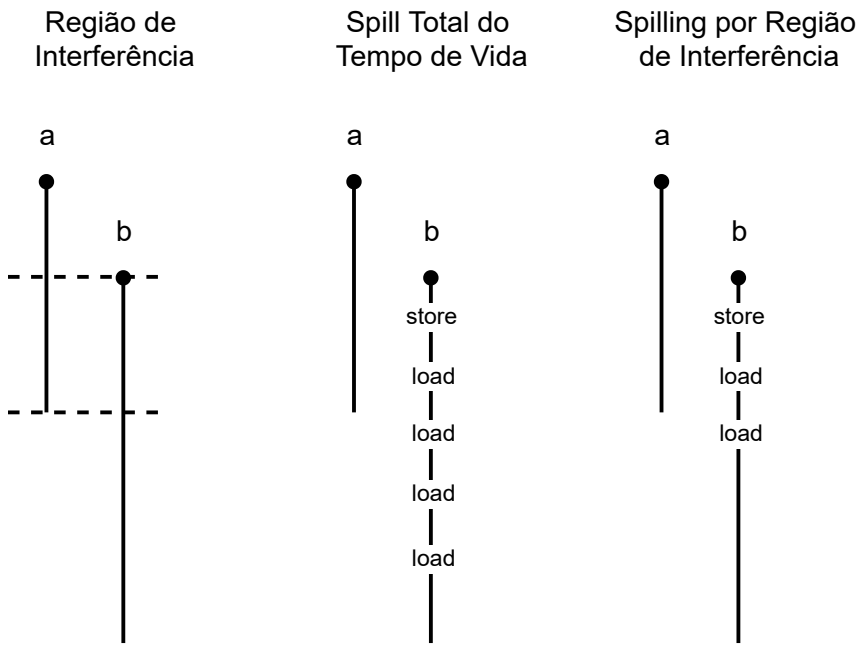


Figura 18 – Ilustração de região de referência. Esta figura foi retirada do trabalho de Bergner *et al.* [3].

Considerando o alocador de Briggs [1], é adicionada uma nova etapa antes de *spill code* para obter o custo de *spill* de regiões de interferência. Essa fase tenta escolher a região de interferência que minimiza a quantidade necessária de *spill*. Ainda, é avaliado se é menos custoso realizar *spill* total do tempo de vida. A eliminação de regiões de interferência podem resultar no acréscimo de operações *load* que não seriam adicionadas no *spill* total. Neste caso é necessário verificar o custo do uso do tempo de vida fora da região de interferência e decidir a opção mais eficiente [3].

Bergner *et al.* [3] observaram que, dentre os *benchmarks* utilizados, *spilling* por região de interferência resultou em uma redução média de 33,6% da quantidade de código de *spill* inserido e de 8,3% no tempo de execução dos programas compilados.

3.4.1 Exemplo de Spilling por Região de Interferência

O exemplo de *spilling* por região de interferência é feito considerando dois registradores e o seguinte código:

```
a = 10;
b = a + 1;
if (a) {
    c = a + 2;
    b = a + c;
    if (c) {
```

```
        b = b + c;
        C = b + c;
    }
    a = b + c;
}
d = a + b;
```

Observá-se que a, b e c estão vivos simultaneamente. Como estão sendo considerados apenas dois registradores, é necessário escolher uma variável para *spill* com base em seus custos. Supondo que a decisão é feita de acordo com o número de instruções de *spill* inseridas, a é escolhido. O *spill* de a exige adição de quatro instruções, enquanto b e c precisam de seis. O seguinte código apresenta o resultado do *spill* completo de a.

```
a = 10;
b = a + 1;
store a;
if (a) {
    load a;
    c = a + 2;
    b = a + c;
    if (c) {
        b = b + c;
        C = b + c;
    }
    a = b + c;
    store a;
}
load a;
d = a + b;
```

Para realizar o *spilling* por região de interferência, é necessário escolher uma das regiões de interferência de a para ser eliminada. A interferência entre a e b exige a utilização de duas instruções *store* e duas instruções *load*, obtendo o mesmo resultado que o *spill* total de a. Por outro lado, a interferência entre a e c precisa de apenas uma instrução *store* e uma *load*. Sendo assim, a região de interferência entre a e c é escolhida para sofrer *spill*. O código resultante fica na forma:

```
a = 10;
b = a + 1;
store a;
if (a) {
    load a;
```

```
    c = a + 2;
    b = a + c;
    if (c) {
        b = b + c;
        C = b + c;
    }
    a = b + c;
}
d = a + b;
```

3.5 Live Range Splitting

Cooper *et al.* [4] propuseram uma técnica para reduzir a inserção de código de *spill*, na qual tempos de vida são divididos em volta de outros, eliminando interferências. Se o espaço entre dois usos de uma variável contêm um tempo de vida de outra, é possível realizar uma divisão em seu tempo de vida para evitar a necessidade de *spill*. A divisão é realizada através do armazenamento temporário na memória entre as duas utilizações. A Figura 19 apresenta uma ilustração de *live range splitting*.

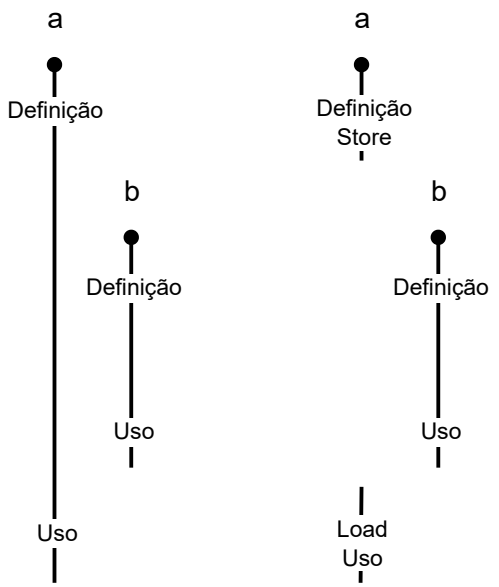


Figura 19 – Ilustração de *live range splitting*.

A divisão de tempos de vida foi utilizada em trabalhos anteriores [28, 15] para diminuir a quantidade de *spill* necessário. Entretanto, essas abordagens são agressivas e possuem poucos mecanismos para tomar boas decisões de divisões. Por causa disso, esses métodos podiam resultar em perda de desempenho em alguns casos [4].

A abordagem de Cooper *et al.* [4] adota uma postura passiva para evitar esse problema. A divisão apenas é considerada após algum tempo de vida ser selecionado para *spill*. Considerando um tempo de vida a, a divisão ocorre se todos os tempos de vida que estão armazenados no mesmo registrador e interferem com a podem se divididas em volta de a, ou se a pode ser dividido em volta deles. Se uma das condições forem atendidas e o custo de divisão for menor que o de *spill*, o processo de *live range split* é realizado.

Para verificar a possibilidade de dividir um tempo de vida em volta de outro, Cooper *et al.* [4] introduziram o *containment graph*. O grafo é similar ao grafo de interferência, mas é direcionado em vez de não direcionado. Uma aresta existe caso pelo menos uma instrução de uso ou definição do vértice de destino está dentro do tempo de vida do vértice de origem. A Figura 20 apresenta exemplos de *containment graphs*.

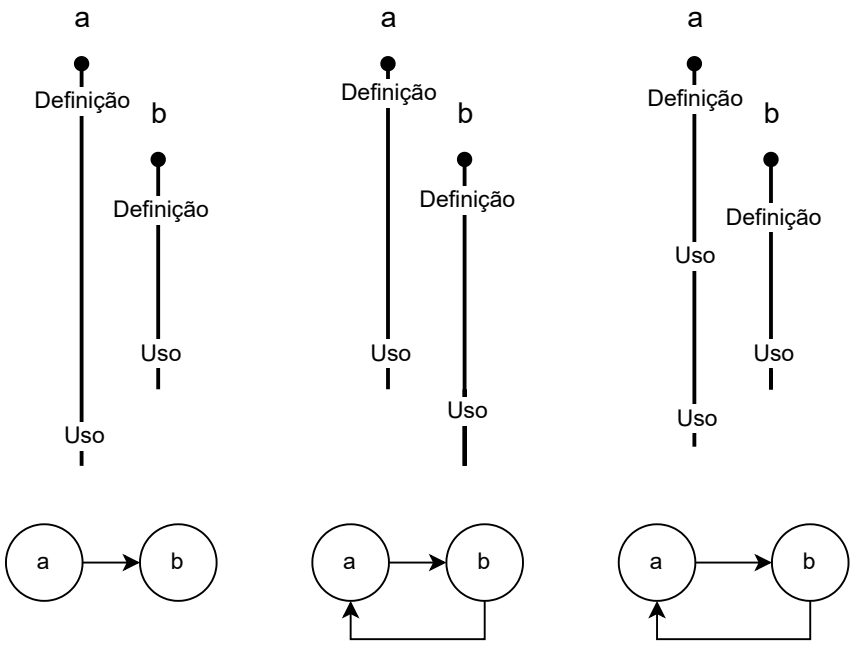


Figura 20 – Exemplo de *containment graphs*. Esta figura foi retirada do trabalho de Cooper *et al.* [4].

Através dos *containment graphs* é possível identificar oportunidade para dividir tempos de vida em volta do outro. Considerando dois vértices a e b:

- Se existe apenas uma aresta de a para b é possível dividir b em volta de a;
- Se existe apenas uma aresta de b para a é possível dividir a em volta de b;
- Se existe uma aresta de a para b e uma aresta de b para a não existe oportunidade para divisão.

Cooper *et al.* [4] testaram sua técnica com 32 *benchmarks*, apresentando resultados positivos. Comparado ao alocador de Briggs, conseguiu reduzir a quantidade de código

de *spill* para a maioria dos casos, chegando a 78% em alguns. Apenas dois *benchmarks* apresentaram maior inserção de código de *spill*. Porém, comparado ao *spilling* por região de interferência (seção 3.4), obteve resultados piores para mais da metade dos casos. Cooper *et al.* apontam a possibilidade de usar ambas abordagens, tomando decisões com bases no custo estimado entre *live range splitting*, *spilling* por região de interferência e *spill* total.

3.5.1 Exemplo de Live Range Splitting

Considerando o seguinte código:

```
a = 5;
b = 0;
c = 10;
while(b < 10){
    b = b + 1;
    print(c)
}
a = a + b;
print(a);
```

O seu *Containment graphs* é apresentado na Figura 21.

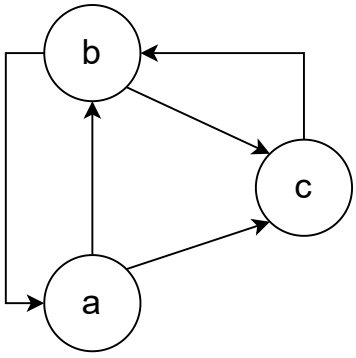


Figura 21 – *Containment graphs* do exemplo.

Suponha que é necessário realizar *spill* em *c*. O resultado do *spill* total de *c* fica na forma:

```
a = 5;
b = 0;
c = 10;
store c;
while(b < 10){
    b = b + 1;
```

```
        load c;
        print(c);
    }
    a = a + b;
    print(a);
```

Como a e b interferem, são alocados registradores diferentes para cada. Então, é necessário verificar a possibilidade de *live range splitting* de a, b e c. Analisando a Figura 21, apenas é possível dividir a em volta de c. Em seguida é preciso comparar o custo entre a divisão de a e *spill* total. Ambos precisam de adição de duas instruções, mas como no *spill* total uma deles está em um *loop*, a opção menos custosa é a divisão de a. Assim, é obtido o seguinte código:

```
    a = 5;
    store a;
    b = 0;
    c = 10;
    while(b < 10){
        b = b + 1;
        print(c);
    }
    load a;
    a = a + b;
    print(a);
```

4 MACHINE LEARNING

Este capítulo apresenta uma fundamentação teórica sobre as técnicas de *machine learning* (aprendizado de máquina), redes neurais e *deep learning* (aprendizado profundo), além de suas formas de treinamento. Apesar que muitas vezes esses termos serem utilizados como sinônimos, cada um se refere a uma subárea diferente dentro da área da inteligência artificial (IA), conforme ilustrado na Figura 22. Assim, *machine learning* é um ramo da inteligência Artificial, redes neurais são uma forma de *machine learning* e *deep learning* é um tipo de rede neural.

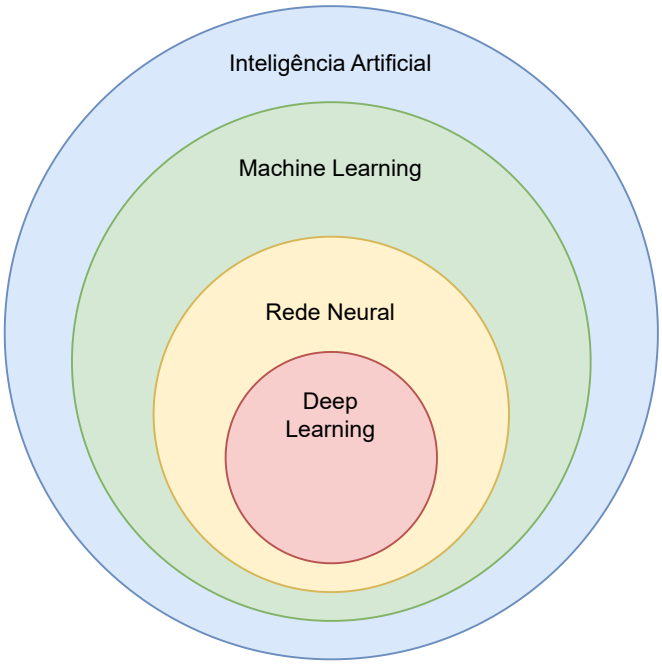


Figura 22 – Inteligência artificial e suas subáreas.

Machine learning, especificamente, é um ramo da inteligência artificial que tem como objetivo o desenvolvimento de algoritmos que possam emular a inteligência humana, tomando decisões e reconhecendo padrões com base em experiências acumuladas através da solução bem sucedida de problemas anteriores [18, 19].

Por meio do uso de métodos estatísticos, os algoritmos são treinados, geralmente com grande quantidade de dados, para fazer classificações ou previsões de acordo com o problema proposto. Essas previsões conduzem a tomada de decisões, de forma a melhorar os resultados do algoritmo [47].

Machine learning pode ser utilizado para diversas tarefas, entre as mais comuns estão [48]:

- **Classificação:** Consiste em atribuir uma categoria para um item presente em uma entrada de dados, como identificar a espécie de um animal em uma imagem;
- **Regressão:** Uma tarefa na qual é necessário prever um valor contínuo, como preço, idade, salário, entre outros;
- ***Ranking*:** Um problema no qual é preciso aprender a ordenar itens de acordo com algum critério. Um exemplo comum deste problema é retornar páginas *web* relevantes de acordo com uma pesquisa;
- ***Clustering*:** Consiste em encontrar padrões de similaridade dentre estes dados de entrada para dividi-los em grupos (*clusters*), procurando ao máximo aumentar a similaridade entre os elementos de um mesmo *cluster* [49].

4.1 Tipos de Aprendizado

Existem várias formas de treinar modelos de *machine learning*, e dependendo do problema e dos dados disponíveis, diferentes tipos de cenários de aprendizado podem ser utilizados. Algum dos tipos de aprendizado mais comuns são [48]:

- **Aprendizado supervisionado:** Neste cenário os conjuntos de dados utilizados no treinamento são rotulados, permitindo que o modelo verifique a resposta correta para medir sua precisão;
- **Aprendizado não supervisionado:** Neste tipo de treinamento o modelo de *machine learning* recebe exclusivamente dados não rotulados. Devido à falta de rótulos se torna mais difícil avaliar o desempenho do modelo. Geralmente é utilizado em problemas de *clustering*;
- **Aprendizado semi supervisionado:** Neste cenário apenas partes dos dados utilizados no treinamento são rotulados. Muitas vezes rotular dados pode ser uma tarefa demorada e custosa, tornando este tipo de aprendizado vantajoso;
- **Reinforcement learning:** Neste cenário o modelo aprende através de *feedbacks* de suas ações. Este cenário é explicado detalhadamente na próxima subseção.

4.1.1 Reinforcement learning

O *reinforcement learning* é uma forma de aprendizado bastante utilizado para problemas mais complexos. A maioria das abordagens de alocação de registradores baseadas

em *machine learning* apresentadas no próximo capítulo utilizam esta forma de aprendizado. Nesse cenário o problema é resolvido por um agente, que recebe o estado atual do problema e interage com ele por meio de uma ação. O agente deve aprender o melhor comportamento para resolver o problema por tentativa e erro em um ambiente dinâmico [50].

No *reinforcement learning* o treinamento e a etapa de teste do modelo de *machine learning* ocorrem simultaneamente. Para coletar informação e aprender, o modelo interage ou afeta o ambiente, recebendo uma recompensa para cada ação. O objetivo do modelo é maximizar a recompensa durante as interações com o ambiente [48].

No *reinforcement learning* o agente enfrenta o dilema *exploration versus exploitation*, tendo que decidir na interação entre realizar novas ações para obter novas informações (*exploration*) e utilizar as informações já coletadas para coletar recompensas já conhecidas (*exploitation*) [48].

Além disso, aplicações de *reinforcement learning* podem ser modeladas com mais de um agente, sendo chamado de *Multi-agent reinforcement learning* [51]. Existem duas formas de realizar o aprendizado com múltiplos agentes: se os agentes trabalham em conjunto para atingir o objetivo, o aprendizado é considerado cooperativo; se os agentes competem entre si para atingir o objetivo, o aprendizado é considerado competitivo. Além disso, o *Multi-agent reinforcement learning* pode apresentar variações em relação ao ambiente. Os agentes podem agir sequencialmente ou formar uma hierarquia, resultando no *Hierarchical Multi-Agent Reinforcement Learning* [52]. Nessa modelagem, os agentes no topo da hierarquia determinam como os agentes em níveis menores devem agir [51, 7].

4.2 Redes Neurais

As Redes Neurais (Neural Networks) são um tipo de algoritmo de *machine learning* que simula o funcionamento das sinapses do cérebro humano para processar e analisar dados. Diferente de abordagens tradicionais de computação, que utilizam séries de blocos para executar as tarefas, as redes neurais são compostas por redes de nós e arestas. Os nós simulam os neurônios e as arestas simulam as sinapses [53].

O primeiro modelo de rede neural foi proposto pelo neurocientista Warren McCulloch e pelo matemático Walter Pitts em 1943, que propuseram a ideia de um neurônio de lógica de limiar para simular o comportamento de neurônios reais e modelar redes neurais artificiais [54]. Em 1975, Kunihiko Fukushima apresentou o conceito de rede neural multicamadas [55], que continua sendo utilizado até hoje. A Figura 23 apresenta um exemplo de rede neural multicamadas.

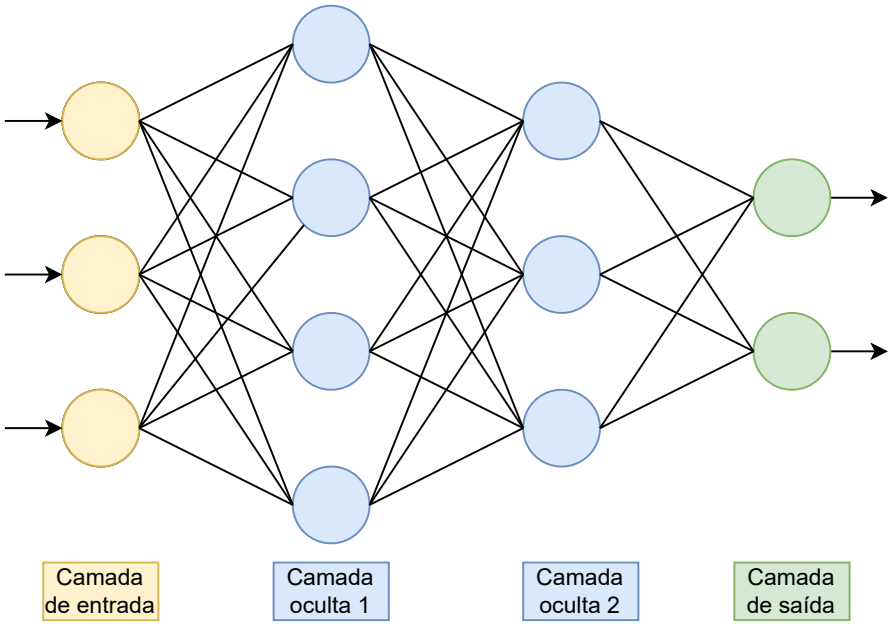


Figura 23 – Exemplo de uma rede neural multicamadas.

Conforme apresentado na Figura 23, as camadas podem ser classificadas em três grupos:

- **Camada de entrada:** Recebe os dados que a rede neural irá analisar;
- **Camada oculta:** São as camadas intermediárias. É onde ocorre a maior parte do processamento;
- **Camada de saída:** É a camada que produz o resultado final ou previsão da rede neural.

Os dados de entrada são processados e passados para a próxima camada através das arestas. Esse processo é repetido até que a camada de saída seja alcançada.

Cada neurônio recebe o produto entre os seus valores de entrada e seus respectivos pesos, os quais serão somados no neurônio. Em seguida é adicionado o *bias* no valor, uma constante que permite transladar o resultado do neurônio. Por fim, o valor calculado é fornecido para uma função de ativação, gerando a saída do neurônio. Esse processo é ilustrado na Figura 24.

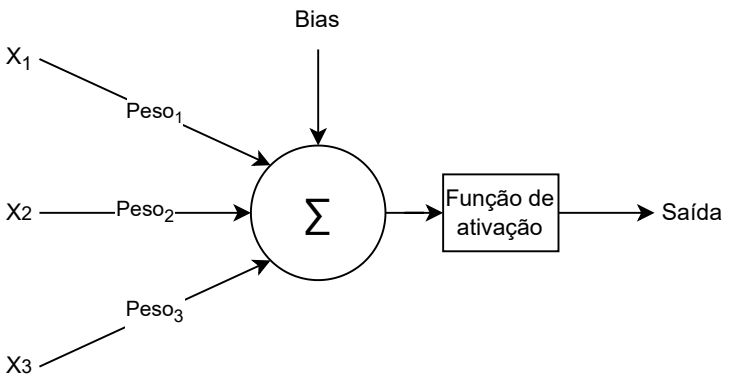


Figura 24 – Operações utilizadas em um neurônio.

O processo de aprendizado em uma rede neural envolve o ajuste dos pesos e *bias* de cada neurônios para minimizar o erro entre a saída da rede neural e o resultado real. O ajuste desses valores é realizado através das técnicas de gradiente descendente e *back-propagation* [56].

4.3 Deep Learning

Deep learning é uma subárea de redes neurais, caracterizada pela maior quantidade de camadas de neurônios. Redes neurais com três ou mais camadas podem ser consideradas algoritmos de *deep learning* [57]. Devido ao maior número de camadas, algoritmos de *deep learning* possuem maior capacidade de processamento, resultando em maior capacidade de abstração e generalização dos dados. Por causa disso, geralmente esses algoritmos são utilizados para problemas mais complexos [58, 59].

Ao contrário de outras formas de *machine learning*, que podem exigir que os dados sejam estruturados e apresentados de maneira específica, o *deep learning* é capaz de aprender a partir de dados brutos, como imagens, áudio e texto, sem a necessidade de pré-processamento de dados. [57].

5 APLICAÇÃO DE MACHINE LEARNING NA ALOCAÇÃO DE REGISTRADORES

Com o avanço das técnicas de *machine learning*, várias aplicações foram propostas em diversas áreas da computação. Porém, devido à falta de dados de treinamento disponíveis, a dificuldade de modelar redes neurais para resolver o problema de alocação de registradores e a necessidade de garantir a corretude da sua solução [7], apenas recentemente foram pesquisadas abordagem de alocação de registradores baseadas em *machine learning*.

Neste contexto, este capítulo apresenta as principais técnicas para resolver o problema de alocação de registradores usando aprendizado de máquina propostas na literatura, assim como trabalhos que, apesar de não abordar o tema diretamente, podem ser úteis para projetar alocadores utilizando *machine learning*. A partir das abordagens apresentadas será possível identificar oportunidades e dificuldades de futuras pesquisas na área.

5.1 Algoritmo de coloração de grafo aproximada baseada em deep learning

Das *et al.* [6] criaram novas heurísticas para algoritmos de alocação de registradores baseados em coloração de grafos. Eles utilizaram *deep learning* para aprender novas heurísticas almejando obter desempenho comparável ou melhor que as heurísticas utilizadas em compiladores modernos.

Para modelar o problema de coloração de grafos foi utilizado um tipo de *Recurrent Neural Network* (RNN), que é uma classe de redes neurais na qual as conexões entre neurônios podem criar um ciclo, permitindo que a saída de alguns nós afete a entrada subsequente para os mesmos nós [60, 61].

Especificamente, foi utilizado o *Long short-term memory* (LSTM), uma variante de RNN. O LSTM é uma arquitetura de *Recurrent Neural Network* que lembra valores em intervalos arbitrários. Usualmente uma unidade LSTM é composta por uma célula que contêm três portões, um para entrada, outra para a saída e outra para "esquecer". Esses portões controlam o fluxo de informação que entra e sai da célula [62]. A Figura 25 apresenta a célula LSTM mais comum e a Figura 26 demonstra o funcionamento de uma *Recurrent Neural Network* LSTM.

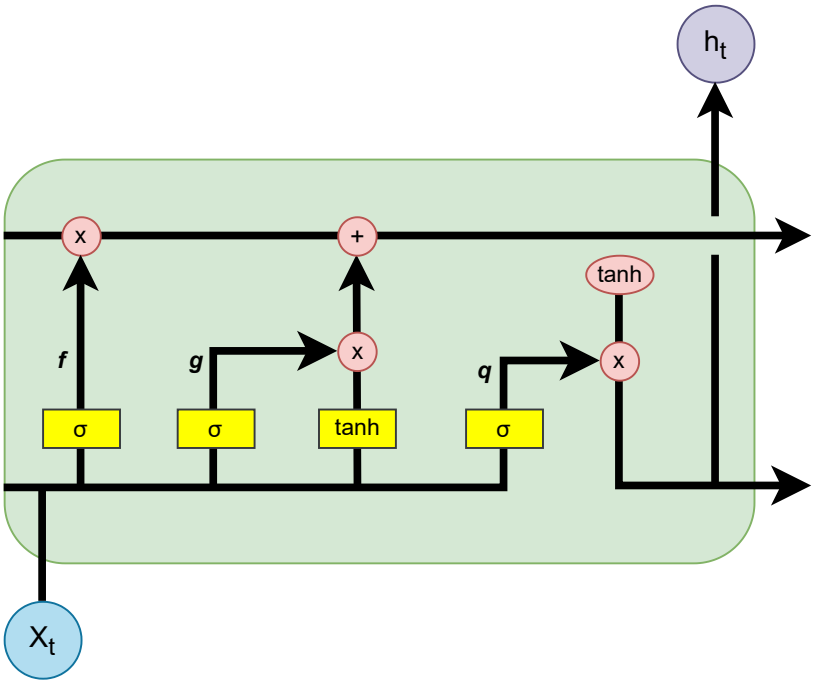


Figura 25 – Exemplo de uma célula LSTM. Esta figura foi retirada do trabalho de Menezes dos Anjos. [5].

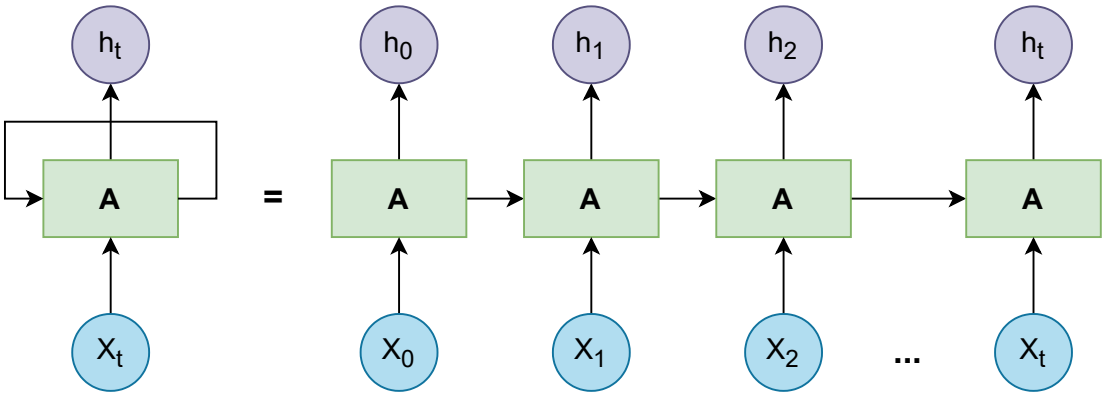


Figura 26 – Exemplo do funcionamento de uma RNN/LSTM. Esta figura foi retirada do trabalho de Das *et al.* [6].

Durante a modelagem da rede neural, Das *et al.* [6] estabeleceram um limite para o tamanho do grafo de interferência, considerando grafos com 100 vértices ou menos. Desta forma, a matriz de adjacência do grafo inteira é utilizada como entrada para a rede neural. A sequência de entrada do LSTM é a sequência dos vértices do grafo. A saída do modelo LSTM são as cores de cada nó. Para melhor a eficiência das previsões foram utilizados *deep learning* LSTM com três camadas. A Figura 27 ilustra essa modelagem.

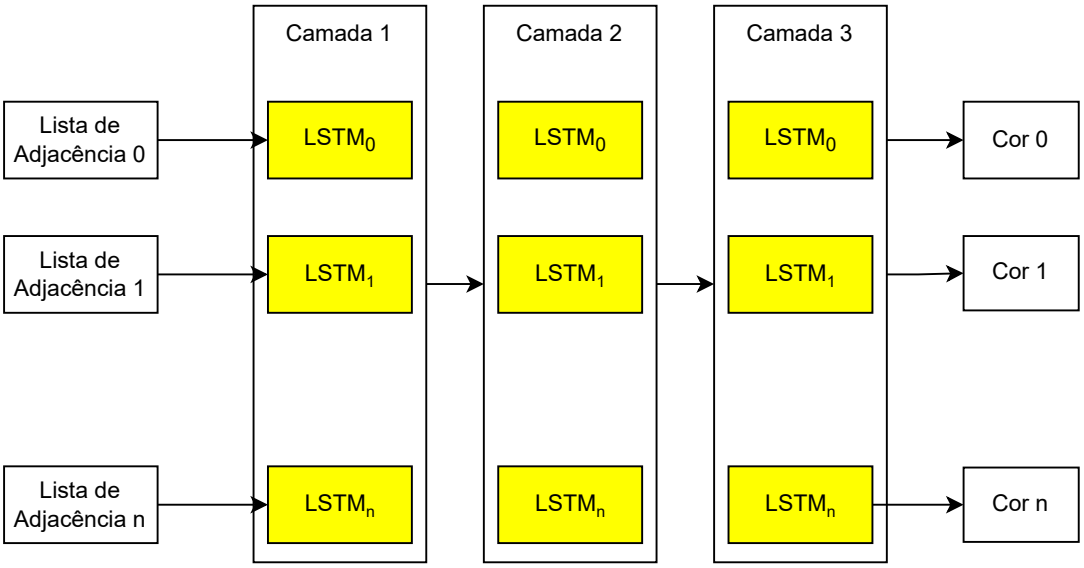


Figura 27 – Ilustração do modelo de Das *et al.* [6].

Após a aplicação da rede neural, é realizada uma verificação no resultado, para garantir que a coloração prevista é válida. Caso dois vértices adjacentes que possuam a mesma cor sejam encontrados, é realizada uma fase de correção de cor para atribuir novas cores paras os vértices com colorações inválidas.

Foram utilizados grafos aleatórios para o treinamento do modelo. Os grafos foram gerados pelo *very nauty* [63], uma biblioteca de C de algoritmos de grafos, especialmente voltada para geração rápida de grafos aleatórios.

Para analisar o seu desempenho e eficácia, a rede neural foi testada com vários grafos de interferência de funções presentes em *benchmarks* do SPEC CPU 2017. Os resultados são comparados com Opção de alocação *Greedy* do LLVM, que utiliza *Linear Scan* com *live range splitting* (Seção 3.5) agressivo. A comparação é realizada de acordo com o número de registradores necessários para obter uma coloração válida sem necessidade de *spill*.

Os dois alocadores foram avaliados em cinco *benchmarks*, nos quais o desempenho foi calculado a partir da soma do número de cores necessárias para cada função. A rede neural apresentou resultados inferiores em apenas um *benchmark*, o *508.namd_r*, com uma diferença de 5%. Para o demais *benchmarks*: *505.mcf_r*, *557.xz_r*, *541.leela_r* e *502.gcc_r*, com resultados melhores em aproximadamente 2%, 2,5%, 7% e 5% respectivamente.

Das *et al.* [6] apontam que o trabalho é um dos primeiros passos na aplicação de *machine learning* para criar novas heurísticas para o problema de alocação de registradores, sendo necessário realizar novas pesquisas sobre o tema para criar modelos mais poderosos.

5.2 Alocação de registradores com Reinforcement Learning

VenkataKeerthy *et al.* [7] criaram uma aplicação baseada em *reinforcement Learning* que realiza completamente a alocação de registradores, além de implementarem o LLVM-gRPC, que permite fácil integração de *machine learning* com o compilador nas fases de treinamento e implantação.

O problema de alocação de registradores foi modelado como um processo de decisão de Markov, que modela tomadas de decisões em ambientes discretos, estocástico e sequenciais [64]. Além disso, foi utilizado *Hierarchical Multi-Agent Reinforcement Learning* [52].

A alocação de registradores foi dividida em quatro tarefas. Cada tarefa possui um agente responsável por ela, sendo que os agentes apresentam níveis de hierarquia diferentes. Após realizar a sua ação o agente invoca um próximo agente de nível inferior. Os agentes são:

- ***Coloring agent***: Agente de nível baixo que aprende a escolher um cor válida para a variável ou realizar *spill* caso não exista registrador disponível. Sua recompensa é definida pelo custo de *spill* da variável. Se o tempo de vida foi colorido então o reforço é positivo, mas se a variável sofre *spill* então o reforço é negativo. Desta forma o agente deve priorizar a coloração de variáveis com maior custo;
- ***Splitter***: Agente de nível baixo que aprende a identificar pontos de divisões dos tempos de vida. Seu objetivo e minimizar o custo de *spill*. Sua recompensa é dada pela diferença entre o custo de *spill* antes e após a divisão;
- ***Task selector***: Agente de nível médio, decide se a variável escolhida é alocada para um registrador ou sofre o processo de *live range splitting* (Seção 3.5). A sua recompensa é definida de acordo com a coloração da variável. Caso a tarefa escolhida seja dividir o tempo de vida então a recompensa é adiada até a decisão de coloração;
- ***Node selector***: Agente com maior nível de hierarquia, responsável por selecionar uma variável para a alocação, determinando a ordem de alocação. Sua função recompensa é calculada de acordo com a decisão final de alocação da variável escolhida.

Os grafos de interferências são obtidos através da *Machine Intermediate Representation* (MIR) do LLVM, uma representação intermediária da etapa de compilação que é humanamente legível [65]. Os grafos são utilizados como entrada para um modelo de *Gated Graph Neural Network* (GGNN) [66], que aprende a gerar a representação de estado que alimenta os agentes *node selector*.

Além disso, VenkataKeerthy *et al.* [7] implementaram a estrutura LLVM-gRPC para integrar o alocador com o compilador LLVM. Essa estrutura fornece comunicação

entre o código em Python e o compilador de C++, oferecendo suporte para o treinamento e implantação do modelo. Para isso, o LLVM-gRPC utiliza chamadas gRPC [67] e a ferramentas do LLVM, aproveitando a estrutura modulada do compilador. A Figura 28 apresenta um diagrama do funcionamento do alocador e sua integração com o LLVM.

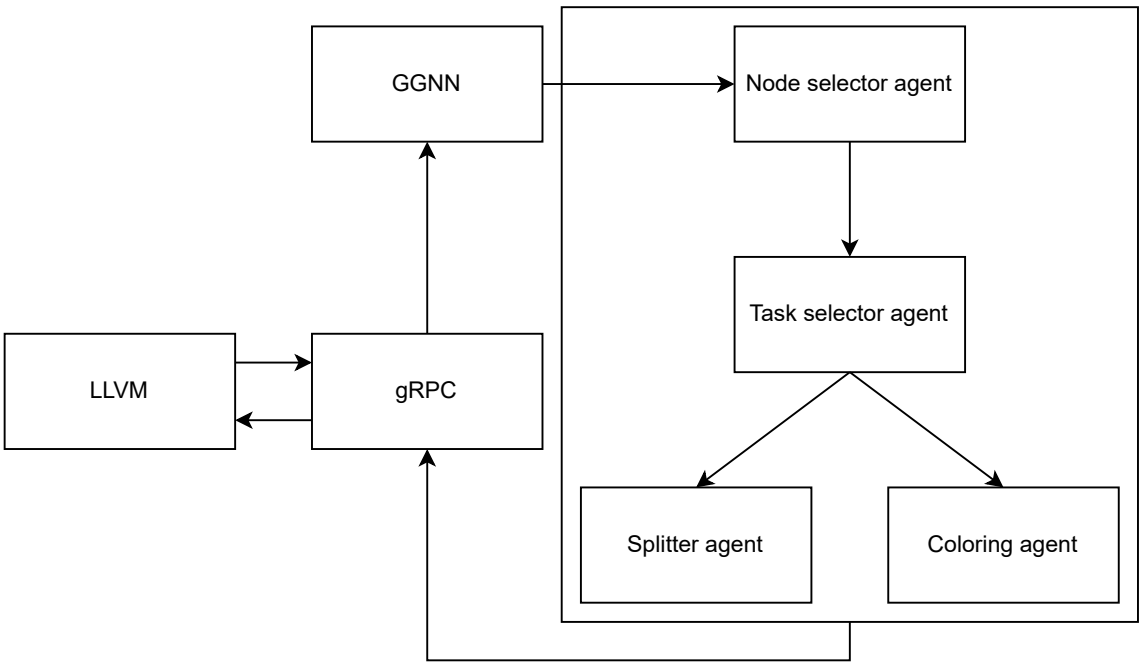


Figura 28 – Ilustração simplificada do funcionamento do alocador de VenkataKeerthy *et al.* [7].

O treinamento foi realizado com 2 mil arquivos coletados aleatoriamente dos *benchmarks* do SPEC CPU 2017 e da biblioteca Boost de C++. Os agentes de *Reinforcement Learning* são treinados utilizando *Proximal Policy Optimization* (PPO) [68]. São treinados dois modelos, um que usa recompensas globais e locais, e outro que utiliza apenas recompensas locais.

Os resultados são avaliados utilizando 18 *benchmarks* do SPEC CPU 2017 e 2006. O modelo desenvolvido é comparado com os alocadores *Greedy*, *Basic* e PBQP do LLVM. Foram coletados tempos de execução em dois processadores: x86 (Intel Xeon SkyLake W2133, 6 cores, 32GB RAM) e AArch64 (ARM Cortex A72, 2 cores, 8GB RAM).

No primeiro processador, o alocador desenvolvido apresentou, em média, um tempo melhor em aproximadamente 18s segundos em relação ao *Basic*, 22s em relação ao PBQP e 4s pior que a opção *Greedy*. Os testes no segundo processador obteve resultados similares, com tempo de execução média melhor em 18s e 13s que o alocador *Basic* e PBQP respectivamente e 1s pior que o *Greedy*.

VenkataKeerthy *et al.* [7] ressalta que o modelo apresenta um desempenho melhor ou comparável com as opções de alocação do LLVM. Além disso, observam que o alocador

desenvolvido na maioria dos *benchmarks* é o melhor ou segundo melhor alocador entre os avaliados.

5.3 Alocação de registradores com PBQP com deep reinforcement learning

Kim *et al.* [9] criaram um alocador voltado para *Automated test equipment* (ATE), um sistema embarcado especial para testar chips de memória DRAM, que apresentam processadores com estrutura altamente irregular. Desta forma, foi proposto uma abordagem de *deep reinforcement learning* para resolver alocação de registradores baseada em PBQP.

O alocador utiliza Monte Carlo *tree search* (MCTS) [69], uma técnica de busca baseada em heurísticas e probabilidades, combinando elementos da implementação clássica do *tree search* com princípios de *reinforcement learning*. Sua principal utilização é em *softwares* para jogar jogos de tabuleiro.

Em muitos problemas é impossível verificar todos os nós das árvores. Por causa disso, algoritmos de *tree search* tradicionais podem ignorar soluções melhores. Por outro lado, algoritmos MCTS avaliam outras alternativas periodicamente, aumentando a possibilidade de encontrar caminhos melhores ao realizar a busca na árvore. Sendo assim, algoritmos MCTS são baseados no dilema *exploration versus exploitation*, explorando novos caminhos de forma balanceada para não aumentar demasiadamente o custo da busca [69].

O Monte Carlo *tree search* é dividido em quatro etapas:

- ***Selection***: O algoritmo percorre a árvore avaliando os nós com base em uma heurística e simulações anteriores;
- ***Expasion***: Um novo nó filho é adicionado no nó selecionado anteriormente;
- ***Simulation***: É realizada uma simulação para determinar ações ou movimentos até que um resultado ou estado pré-determinado seja atingido;
- ***Backpropagation***: Após determinar o valor do novo nó o restante da árvore é adaptada de acordo com o resultado da simulação.

Essas etapas são repetidas até que a solução seja encontrada. A Figura 29 ilustra o funcionamento do Monte Carlo *tree search*.

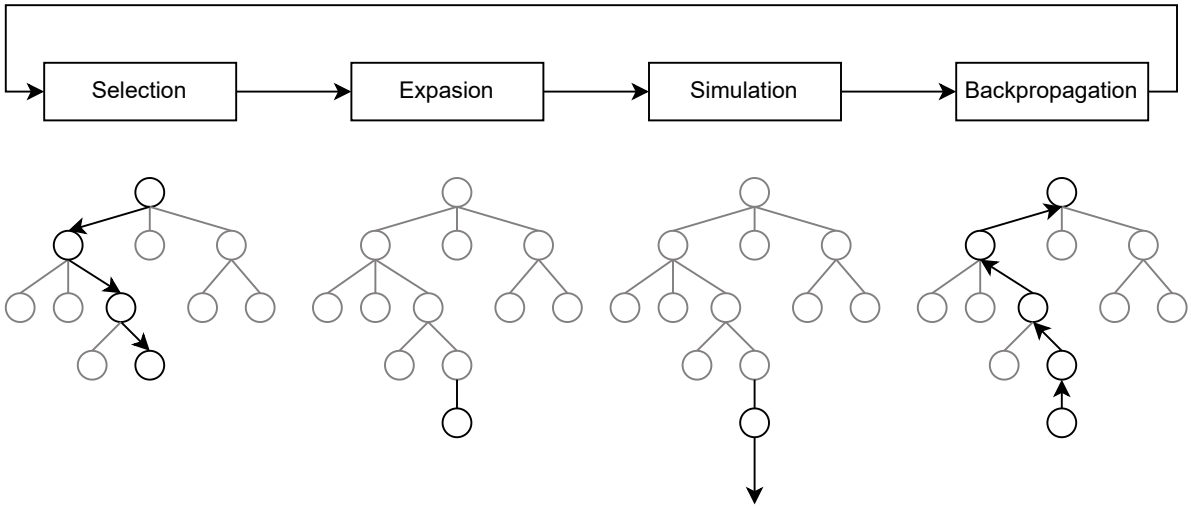


Figura 29 – Ilustração do funcionamento do Monte Carlo *tree search*. Esta figura foi retirada do trabalho de Chaslot *et al.* [8].

Além disso, também foi utilizado o modelo de *Deep Reinforcement Learning* do *AlphaZero*, que foi desenvolvido pela empresa de pesquisa em inteligência artificial DeepMind para dominar os jogos de xadrez, shogi e go [70]. Neste contexto, a alocação de registradores utilizando PBQP é abstraída como um jogo, sendo resolvida através de um algoritmo Monte Carlo *tree search* utilizando a rede neural do *AlphaZero* para obter os resultados das simulações.

Para utilizar o modelo de *Deep Reinforcement Learning* do *AlphaZero* é necessário representar grafos PBQP em forma de vetor numérico. Para obter esta representação, Kim *et al.* [9] utilizou *message passing-based graph convolutional network* (GCN) com modificações [71, 72]. Ainda, foi preciso modular o problema de coloração de grafo PBQP como um jogo. Para isso, é necessário formular *action*, *state* e *reward* para PBQP:

- ***Action***: Colorir o próximo vértice sem cor;
- ***State***: Representação do Grafo em determinado momento da coloração. Apresenta os vértices, arestas, vetores e matrizes de custo e informações como número de cores disponíveis e número de ações anteriores;
- ***Reward***: Indica o resultado do jogo para o treinamento da rede neural. No *AlphaZero* uma vitória é +1, empate 0 e derrota -1. Entretanto, o PBQP seria um jogo com apenas um jogador, assim não é possível julgar o desempenho do jogador (Rede Neural). Para contornar esse problema, o resultado é obtido através da comparação com o melhor desempenho anterior no mesmo jogo (grafo). Sendo assim, uma vitória é obtida ao alcançar um custo de coloração menor em relação às iterações anteriores.

Desta forma, o jogo consiste em uma árvore de *actions*, sendo necessário buscar o melhor caminho de coloração. Assim, a rede neural é responsável pela etapa *simulation* do Monte Carlo *tree search*, calculando a probabilidade de vitória de uma determinada ação. Por fim, foi formulado uma heurística para a etapa *select*, que é obtida através da combinação dos valores da probabilidade de vitória do nó, o *reward* esperado da *action* e o número de vezes que uma ação foi escolhida. A Figura 30 apresenta um diagrama do funcionamento do modelo Kim *et al.*

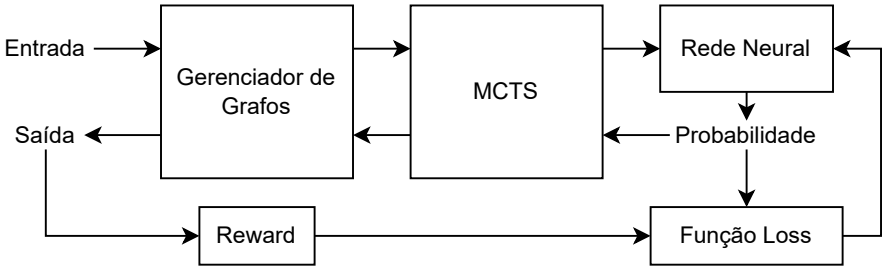


Figura 30 – Ilustração simplificada do modelo de Kim *et al.* [9].

O modelo foi treinado utilizando grafos aleatórios gerados pelo modelo de Erdos-Rényi [73]. Para verificar o desempenho do modelo foram utilizados 24 *benchmarks* presentes no LLVM *test-suite* [74]. O alocador de Kim *et al.* foi comparado com as opções *greedy*, PBQP e *fast* do LLVM. A opção *fast* utiliza alocação de registradores local e foi utilizado como base para os valores coletados. Os três alocadores apresentaram um tempo aproximadamente 1,4 vezes maior que a opção *fast*. Além disso, quando comparado com o alocador PBQP do LLVM, o modelo de Kim *et al.* apresentou soluções piores em apenas dois *benchmarks*.

5.4 Trabalhos relacionados

Apesar de não abordarem diretamente a alocação de registradores, outros trabalhos apresentam aplicações de *machine learning* em contextos similares, principalmente na resolução do problema de coloração de grafos. Neste contexto, é importante conhecer as técnicas utilizadas nessas aplicações para identificar oportunidades para novas pesquisas sobre a utilização de *machine learning* na alocação de registradores.

Schuetz *et al.* [75] desenvolveu um modelo para resolver o problema de coloração de grafos inspirado em conceitos de mecânica estatística. Sua técnica utiliza *Graph neural networks* [76] e o modelo de Potts [77].

O modelo de Potts [77] descreve interações em um reticulado cristalino através de *spins*. O modelo é utilizado para estudar comportamento de ferromagnetos e outros fenômenos físicos no estado sólido. Nesse modelo cada vértice é associado a uma variável

de *spin* que pode assumir um determinado número de estados. Através dos valores de *spin* são simuladas interações de acordo com o fenômeno estudado.

No contexto da aplicação Schuetz *et al.* os estados representam as cores dos vértices, e as regras de iterações forçam que vértices com os mesmos valores de *spin* alterem o valor ao decorrer das interações.

Ainda, Zhou *et al.* [78] desenvolveu uma abordagem de busca local baseada em *reinforcement learning* para resolver problemas de agrupamentos. Para demonstrar a sua utilização, Zhou *et al.* apresentaram uma aplicação para resolver coloração de grafos.

Zhou *et al.* realizaram a busca local baseada em *reinforcement learning* nas possíveis soluções do problema. O algoritmo começa com uma solução aleatória do espaço de busca. Em cada interação o algoritmo busca uma solução vizinha melhor de acordo com uma função de avaliação. A busca é encerrada quando uma coloração válida é obtido ou quando não existem soluções vizinhas melhores, encontrando assim uma solução ótima local.

Além desses trabalhos, é possível encontrar na literatura outras aplicações de *machine learning* para a resolução do problema de coloração de grafos que utilizam técnicas e conceitos pouco exploradas no contexto da alocação de registradores. Por exemplo, o trabalho de Shen *et al.* [79] utiliza conceitos de programação linear para abstrair o problema de coloração de grafo e aplicar *machine learning*. Ainda, a pesquisa de Khakhulin *et al.* [80] utiliza *reinforcement learning* para resolver o problema de decomposição em árvore, o qual ao ser resolvido permite a solução da coloração de grafos em tempo linear.

6 DIREÇÕES PROMISSORAS PARA TRABALHOS FUTUROS

A aplicação de técnicas de *machine learning* em diversas áreas da computação tem sido cada vez mais comum, porém, no contexto da alocação de registradores, poucos trabalhos foram desenvolvidos até o momento. Isso se deve em parte à complexidade do problema e à falta de dados de treinamento disponíveis, o que pode ser observado em alguns trabalhos apresentados no Capítulo 5, que recorreram a grafos gerados aleatoriamente para treinar seus modelos.

No entanto, a alocação de registradores é um desafio importante na compilação de código para processadores modernos, e a aplicação de técnicas de aprendizado de máquina pode trazer benefícios significativos, como melhorias no desempenho e na eficiência energética dos processadores [81, 82, 83]. Sendo assim, é importante explorar as diferentes formas de aplicar o aprendizado de máquina nesse problema e avaliar o desempenho dos modelos propostos em diferentes arquiteturas de processadores e em diferentes tipos de programas.

Nesse contexto, este capítulo tem como objetivo apresentar algumas direções promissoras para novas pesquisas sobre a aplicação de *machine learning* na alocação de registradores. Com base nas técnicas apresentadas no Capítulo 5 serão discutidos os principais desafios e oportunidades de pesquisa nessa área. Espera-se, assim, contribuir para o desenvolvimento de modelos mais eficientes e adaptáveis às necessidades dos programas, possibilitando melhorias significativas no desempenho e na eficiência energética dos processadores.

6.1 Coloração de grafos

Conforme apresentado no Capítulo 5, existem várias formas de resolver a coloração de grafos através de *machine learning*. Os modelos de VenkataKeerthy *et al.* (Seção 5.2) e de Das *et al.* (Seção 5.1) apresentam propostas bastante diferentes para resolver o problema de alocação de registradores. Além disso, também é possível observar na Seção 5.4 outras abordagens para resolver a coloração de grafos utilizando *machine learning*, mesmo que fora do contexto da alocação de registradores.

Neste contexto, uma linha de pesquisa interessante é explorar as diferentes formas de aplicar *machine learning* na alocação de registradores, além de analisar como diferentes modelagens afetam o desempenho do alocador. Sendo assim, trabalhos futuros podem adaptar modelos como os descritos na Seção 5.4 ou criar novas abstrações para resolver o problema.

Por exemplo, um tema interessante para uma nova pesquisa é o desenvolvimento de um alocador baseado no trabalho de Schuetz *et al.* apresentado na Seção 5.4, que utiliza o modelo de Potts [77] para resolver o problema de coloração de grafos.

6.2 PBQP

Assim como na coloração de grafos, a alocação de registradores baseada em PBQP pode ser resolvida utilizando diversas abordagens, apresentando potencial para futuras pesquisas. Porém, diferente da coloração de grafo, existem poucas técnicas na literatura para resolver o PBQP com *machine learning*. Sendo assim, trabalhos futuros podem encontrar mais obstáculos, visto que o tema foi pouco explorado na literatura e não existem muitos modelos para utilizar com base ou inspiração.

Desta forma, é necessário criar novos modelos para resolver o PBQP através de *machine learning*. O trabalho de Kim *et al.* descrito na Seção 5.3 formulou o problema como um jogo, utilizando o algoritmo de busca Monte Carlo *tree search* e uma rede neural para calcular a probabilidade de vitória de uma jogada. Apesar de não ser uma abstração comumente utilizada em métodos tradicionais para resolver o problema, este modelo apresenta uma estrutura similar a alguns algoritmos PBQP, utilizando algoritmos de busca para decidir as melhores ações durante a resolução do problema.

Portanto, um caminho interessante para novos trabalhos sobre o tema é desenvolver modelos similares ao apresentado por Kim *et al.* baseados em algoritmos de busca utilizados em algoritmos PBQP tradicionais. Alterando o método de busca, a rede neural passa a desempenhar um papel diferente no alocador, realizando novos tipos de previsões. A criação de novas modelagens permitem o estudo de diferentes formas de aplicar *machine learning* no problema. Desta forma é possível verificar quais são os métodos mais eficientes para utilizar *machine learning* no PBQP e na alocação de registradores.

Por exemplo, um algoritmo de busca que pode ser utilizado é o *branch-and-bound*. Conforme mencionado na Seção 2.3, *branch-and-bound* é uma técnica bastante utilizada em algoritmo PBQP para alocação de registradores. O algoritmo gera uma árvore de sub-grafos, sendo que cada possível redução é avaliada de acordo com uma heurística, permitindo a eliminação de certos caminhos para manter a complexidade do algoritmo viável. Neste contexto, é possível modelar uma rede neural para ser responsável pelas decisões de *Bounding*, escolhendo as melhores reduções e tomando decisões de alocações e *spill*. Assim, é possível o desenvolvimento de um projeto interessante sobre o tema.

6.3 Minimização de spill

Os métodos de minimização de *spill* são importantes para aproveitar ao máximo a utilização de registradores. No entanto, como a aplicação de *machine learning* na alocação de registradores ainda é um tema pouco explorado na literatura, menos ainda foi pesquisado sobre sua utilização em técnicas de minimização de *spill*. Sendo assim, estudos sobre algoritmos de minimização de *spill* com *machine learning* podem resultar em trabalhos inovadores.

Considerando o levantamento bibliográfico realizado neste trabalho, apenas VenkataKeerthy *et al.* desenvolveu uma aplicação de *machine learning* para minimizar *spill*. Dentro de seu modelo, um agente é responsável para decidir entre alocar ou realizar *live range splitting* (Seção 3.5) nas variáveis. Além disso, outro agente é responsável pela divisão dos tempos de vida, conforme descrito na Seção 5.2.

Similarmente, poderiam ser utilizada estruturas parecidas para aplicar outros métodos de minimização de *spill* utilizando aprendizado de máquina, como *rematerialization* (Seção 3.3) e *spilling* por região de interferência (Seção 3.4).

Conforme descrito na Seção 3.3, um dos principais desafios da *rematerialization* é identificar variáveis adequadas para o processo. Avaliar se a *rematerialization* resulta em um custo maior ou menor que *spill* é um processo complexo e em certos casos ineficiente. Sendo assim, utilizar *machine learning* para prever oportunidades de *rematerialization* pode resultar em uma melhor avaliação das variáveis e, consequentemente, maior minimização de *spill*.

Ainda, *spilling* por região de interferência poderia se beneficiar da utilização de *machine learning* de maneira semelhante do *live range splitting*. A utilização de modelos de aprendizado de máquina para decidir em quais casos é melhor aplicar *spilling* por região de interferência ou apenas deixar a variável sofrer *spill* pode resultar em um melhor aproveitamento do método. Além disso, outra possível aplicação de *machine learning* no *spilling* por região de interferência é a utilização de redes neurais para decidir quais regiões de interferência devem ser eliminadas. Identificar as melhores regiões para sofrer *spill* pode resultar em maior minimização de custo de *spill*.

6.4 Datasets

Uma das maiores dificuldades enfrentada por pesquisas nessa área é a falta de um conjunto de dados apropriado para o treinamento de modelos *machine learning* aplicados na alocação de registradores. Até o momento não existem *datasets* voltados para esses modelos. Por causa disso, Das *et al.* 5.1 e Kim *et al.* 5.3 recorreram a grafos gerados aleatoriamente para treinar seu modelo.

No entanto, grafos aleatórios podem não simular fielmente a característica de diferentes programas, limitando assim o potencial de aprendizado de modelos de *machine learning*. Sendo assim, projetos para gerar *datasets* para o treinamento de modelos *machine learning* aplicados na alocação de registradores podem apresentar um grande impacto na área, facilitando e incentivando novos trabalhos sobre o tema.

Para montar um conjunto de dado, é importante o considerando contexto em que o modelo a ser treinado será utilizado. Dependendo do objetivo da aplicação, o *dataset* precisa suprir diferentes necessidades. Para alocadores de uso geral é importante coletar uma grande variedade de dados, contendo códigos bastante diversos e representando diferentes arquiteturas. Assim o modelo conseguirá generalizar para diferentes tipos de programas e arquiteturas.

Por outro lado, caso o compilador seja destinado para aplicações ou arquiteturas específicas, é necessário coletar dados e códigos considerando o contexto no qual o alocador será utilizado. A inclusão de dados e códigos de outras aplicações ou arquiteturas evita que o modelo considere apenas o contexto no qual será utilizado, afetando negativamente o aprendizado da rede neural.

Por fim, também é importante considerar a abstração do problema para decidir a melhor representação dos dados. Dependendo da modelagem do alocador, o conjunto de treinamentos poderia conter os próprios códigos fontes dos programas, dados sobre os tempos de vidas, grafos de interferência ou grafos PBQP.

6.5 Heurísticas para decisões de spill

As técnicas apresentadas na Seção 5 utilizam *machine learning* para realizar a alocação de registradores em si, porém é possível utilizar aprendizado de máquina para auxiliar o processo. A alocação de registradores utiliza várias heurísticas, especialmente para estimar custo de *spill* e definir a prioridade de alocação das variáveis.

Para alocar os registradores da maneira mais eficiente possível é importante considerar diversos fatores, como o número de utilizações da variável, a quantidade de interferências com outros tempos de vida, além de outros fatores. No entanto, balancear estes fatores é uma tarefa complexa, sendo necessária a utilização de heurísticas, que muitas vezes podem ser melhoradas.

Sendo assim, utilizar *machine learning* para obter heurísticas mais eficientes é um caminho promissor para melhorar o desempenho de alocadores. Por exemplo, a aplicação de uma rede neural para prever o melhor tempo de vida para sofrer *spill* pode resultar em alocações mais eficientes e, consequentemente, melhor tempo de execução de programas compilados.

Ainda, é importante considerar que, assim como os *datasets* mencionados na Seção 6.4, heurísticas são desenvolvidas considerando contextos específicos. Desta forma, é importante considerar o contexto da aplicação em trabalhos sobre esse tema. Assim, a geração de novas heurísticas utilizando *machine learning* em diferentes contextos e aplicações oferece diversos caminhos para futuras pesquisas.

7 CONCLUSÃO

A alocação de registradores é um problema complexo com grande impacto na otimização de programas e no aproveitamento do processador. Nesse contexto, a aplicação de técnicas de aprendizado de máquina apresenta potencial para fornecer soluções mais eficientes, precisas e adaptáveis do que as abordagens tradicionais baseadas em heurísticas.

Neste trabalho foi realizado um estudo abrangente sobre a aplicação de *machine learning* na alocação de registradores. Foi apresentado uma profunda fundamentação teórica sobre as técnicas utilizadas no problema de alocação de registradores e minimização de *spill*. Ainda, foram discutidas as principais abordagens presentes na literatura sobre o tema. Por fim, foram indicados oportunidades e dificuldades para futuras pesquisas na área.

Apesar de ser um tema que apenas recentemente foi pesquisado, foi possível encontrar abordagens promissoras sobre a utilização de *machine learning* na alocação de registradores. Além disso, foram apontados diversos caminhos interessantes para futuros trabalhos, incluindo a adaptação de modelos existentes e o desenvolvimento de abordagens híbridas que combinem modelos de aprendizado de máquina com técnicas tradicionais. Ainda, foram discutidas o desenvolvimento de novas heurísticas, técnicas de minimização de *spill* e conjuntos de dados para aplicações de *machine learning* na alocação de registradores.

Desta forma, espera-se que este trabalho forneça uma base e inspiração para novas pesquisas sobre o tema. Permitindo, assim, o desenvolvimento de novas soluções para o problema de alocação de registradores usando *machine learning* e, consequentemente, contribuindo para melhorias significativas no desempenho e na eficiência energética de códigos gerados por compiladores.

REFERÊNCIAS

[1] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 16, n. 3, p. 428–455, may 1994. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/177492.177575>.

[2] BUCHWALD, S.; ZWINKAU, A.; BERSCH, T. Ssa-based register allocation with pbqp. In: KNOOP, J. (Ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 42–61. ISBN 978-3-642-19861-8.

[3] BERGNER, P. et al. Spill code minimization via interference region spilling. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 32, n. 5, p. 287–295, may 1997. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/258916.258941>.

[4] COOPER, K. D.; SIMPSON, L. T. Live range splitting in a graph coloring register allocator. In: KOSKIMIES, K. (Ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 174–187. ISBN 978-3-540-69724-4.

[5] ANJOS, C. E. M. d. Exploração de arquiteturas de redes neurais em uma série temporal financeira. Universidade Federal do Rio de Janeiro, 2018.

[6] DAS, D.; AHMAD, S. A.; KUMAR, V. Deep learning-based approximate graph-coloring algorithm for register allocation. In: *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. [S.l.: s.n.], 2020. p. 23–32.

[7] VENKATAKEERTHY, S. et al. *RL4ReAl: Reinforcement Learning for Register Allocation*. arXiv, 2022. Disponível em: <https://arxiv.org/abs/2204.02013>.

[8] CHASLOT, G.; WINANDS, M.; HERIK, H. Parallel monte-carlo tree search. In: . [S.l.: s.n.], 2008. p. 60–71. ISBN 978-3-540-87607-6.

[9] KIM, M.; PARK, J.-K.; MOON, S.-M. Solving pbqp-based register allocation using deep reinforcement learning. In: IEEE. *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [S.l.], 2022. p. 1–12.

[10] SILVA, F. L. da. *Color flipping : minimização de spill code via troca de cores em um grafo de interferência*. 2015.

[11] PEREIRA, F. M. Q. *A survey on register allocation*. [S.l.], 2008.

[12] HARTMANIS, J. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, Society for Industrial and Applied Mathematics, v. 24, n. 1, p. 90, 1982.

[13] BOUCHEZ, F. et al. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In: SPRINGER. *International Workshop on Languages and Compilers for Parallel Computing*. [S.l.], 2006. p. 283–298.

[14] CHAITIN, G. J. et al. Register allocation via coloring. *Computer Languages*, v. 6, n. 1, p. 47–57, 1981. ISSN 0096-0551. Disponível em: <https://www.sciencedirect.com/science/article/pii/0096055181900485>.

[15] CHOW, F. C.; HENNESSY, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 12, n. 4, p. 501–536, oct 1990. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/88616.88621>.

[16] CHEN, W.-Y. et al. Register allocation for intel processor graphics. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2018. (CGO 2018), p. 352–364. ISBN 9781450356176. Disponível em: <https://doi.org/10.1145/3168806>.

[17] MOSANER, R. Machine learning to ease understanding of data driven compiler optimizations. In: *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. New York, NY, USA: Association for Computing Machinery, 2020. (SPLASH Companion 2020), p. 4–6. ISBN 9781450381796. Disponível em: <https://doi.org/10.1145/3426430.3429451>.

[18] MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. *Sistemas inteligentes-Fundamentos e aplicações*, Manole, v. 1, n. 1, p. 32, 2003.

[19] NAQA, I. E.; MURPHY, M. J. What is machine learning? In: NAQA, I. E.; LI, R.; MURPHY, M. J. (Ed.). *Machine Learning in Radiation Oncology: Theory and Applicationso*. Cham: Springer, 2015. p. 3–11. ISBN 978-3-319-18305-3. Disponível em: <https://doi.org/10.1007/978-3-319-18305-3_1>.

[20] FURSIN, G. et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, Springer, v. 39, n. 3, p. 296–327, 2011.

[21] ASHOURI, A. H. et al. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM New York, NY, USA, v. 14, n. 3, p. 1–28, 2017.

[22] MENDIS, C. et al. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: PMLR. *International Conference on machine learning*. [S.l.], 2019. p. 4505–4515.

[23] KULKARNI, S. et al. Automatic construction of inlining heuristics using machine learning. In: IEEE. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [S.l.], 2013. p. 1–12.

[24] STEPHENSON, M. et al. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices*, ACM New York, NY, USA, v. 38, n. 5, p. 77–90, 2003.

[25] LEMOS, H. et al. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. [S.l.: s.n.], 2019. p. 879–885.

[26] SAHA, B. K. et al. A machine learning approach to automatic creation of architecture-sensitive performance heuristics. In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. [S.l.: s.n.], 2017. p. 18–25.

[27] COOPER, K.; TORCZON, L. *Engineering a Compiler: International Student Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 9780080472676.

[28] BRIGGS, P. *Register Allocation via Graph Coloring*. Tese (Doutorado), 1992.

[29] POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 5, p. 895–913, sep 1999. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/330249.330250>.

[30] SCHOLZ, B.; ECKSTEIN, E. Register allocation for irregular architectures. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 7, p. 139–148, jun 2002. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/566225.513854>.

[31] BUCHWALD, S.; ZWINKAU, A. Instruction selection by graph transformation. In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2010. (CASES '10), p. 31–40. ISBN 9781605589039. Disponível em: <https://doi.org/10.1145/1878921.1878926>.

[32] ECKSTEIN, E.; KÖNIG, O.; SCHOLZ, B. Code instruction selection based on ssa-graphs. In: KRALL, A. (Ed.). *Software and Compilers for Embedded Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 49–65. ISBN 978-3-540-39920-9.

[33] HAMES, L.; SCHOLZ, B. Nearly optimal register allocation with pbqp. In: LIGHTFOOT, D. E.; SZYPERSKI, C. (Ed.). *Modular Programming Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 346–361. ISBN 978-3-540-40928-1.

[34] LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. Washington, DC, USA: IEEE Computer Society, 2004. (CGO '04), p. 75–. ISBN 0-7695-2102-9. Disponível em: <http://dl.acm.org/citation.cfm?id=977395.977673>.

[35] HSU, W.-C.; FISCHER, C.; GOODMAN, J. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, v. 15, n. 10, p. 1252–1260, 1989.

[36] FARACH-COLTON, M.; LIBERATORE, V. On local register allocation. *Journal of Algorithms*, v. 37, n. 1, p. 37–65, 2000. ISSN 0196-6774. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0196677400910952>.

[37] CALLAHAN, D.; KOBLENZ, B. Register allocation via hierarchical graph coloring. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 26, n. 6, p. 192–203, may 1991. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/113446.113462>.

[38] HENDREN, L. J. et al. A register allocation framework based on hierarchical cyclic interval graphs. In: *Proceedings of the 4th International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 1992. (CC '92), p. 176–191. ISBN 3540559841.

[39] PROEBSTING, T. A.; FISCHER, C. N. Demand-driven register allocation. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 6, p. 683–710, nov 1996. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/236114.236117>.

[40] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 24, n. 7, p. 258–263, jun 1989. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/74818.74841>.

[41] KOSEKI, A.; KOMATSU, H.; NAKATANI, T. Spill code minimization by spill code motion. In: *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. [S.l.: s.n.], 2003. p. 125–134.

[42] CHAITIN, G. J. Register allocation & spilling via graph coloring. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: ACM, 1982. (SIGPLAN '82), p. 98–105. ISBN 0-89791-074-5. Disponível em: <http://doi.acm.org/10.1145/800230.806984>.

[43] SERRANO, M. Inline expansion: When and how? In: GLASER, H.; HARTEL, P.; KUCHEN, H. (Ed.). *Programming Languages: Implementations, Logics, and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 143–157. ISBN 978-3-540-69537-0.

[44] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Rematerialization. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 27, n. 7, p. 311–321, jul 1992. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/143103.143143>.

[45] CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 13, n. 4, p. 451–490, oct 1991. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/115372.115320>.

[46] WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 13, n. 2, p. 181–210, apr 1991. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/103135.103136>.

[47] O que é machine learning? Https://www.ibm.com/br-pt/cloud/learn/machine-learning. Accessed: 2022-13-08.

[48] MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. *Foundations of machine learning*. [S.l.: s.n.], 2018.

[49] LIKAS, A.; VLASSIS, N.; J. Verbeek, J. The global k-means clustering algorithm. *Pattern Recognition*, v. 36, n. 2, p. 451–461, 2003. ISSN 0031-3203. Biometrics. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0031320302000602>.

[50] KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, v. 4, p. 237–285, 1996.

[51] BUŞONIU, L.; BABUŠKA, R.; SCHUTTER, B. D. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, Springer, p. 183–221, 2010.

[52] MAKAR, R.; MAHADEVAN, S.; GHAVAMZADEH, M. Hierarchical multi-agent reinforcement learning. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. New York, NY, USA: Association for Computing Machinery, 2001. (AGENTS '01), p. 246–253. ISBN 158113326X. Disponível em: <https://doi.org/10.1145/375735.376302>.

[53] WANG, S.-C.; WANG, S.-C. Artificial neural network. *Interdisciplinary computing in java programming*, Springer, p. 81–100, 2003.

[54] MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

[55] FUKUSHIMA, K. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, Springer, v. 20, n. 3, p. 121–136, 1975.

[56] LIPPMANN, R. An introduction to computing with neural nets. *IEEE Assp magazine*, IEEE, v. 4, n. 2, p. 4–22, 1987.

[57] WHAT is deep learning? Https://www.ibm.com/topics/deep-learning. Accessed: 2023-16-03.

[58] SCHMIDHUBER, J. Deep learning in neural networks: An overview. *Neural Networks*, v. 61, p. 85–117, 2015. ISSN 0893-6080. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0893608014002135>.

[59] LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group UK London, v. 521, n. 7553, p. 436–444, 2015.

[60] GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep learning*. [S.l.]: MIT press, 2016.

[61] MEDSKER, L. R.; JAIN, L. Recurrent neural networks. *Design and Applications*, v. 5, p. 64–67, 2001.

[62] HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural Computation*, v. 9, n. 8, p. 1735–1780, 1997.

[63] BRIGGS, K. *The very_nauty graph library.* Http://keithbriggs.info/very_nauty.html. Accessed: 2023-24-03.

[64] LITTMAN, M. Markov decision processes. In: SMELSER, N. J.; BALTES, P. B. (Ed.). *International Encyclopedia of the Social & Behavioral Sciences.* Oxford: Pergamon, 2001. p. 9240–9242. ISBN 978-0-08-043076-8. Disponível em: <https://www.sciencedirect.com/science/article/pii/B0080430767006148>.

[65] MACHINE IR (MIR) Format Reference Manual. Https://llvm.org/docs/MIRLangRef.html. Accessed: 2023-27-03.

[66] LI, Y. et al. *Gated Graph Sequence Neural Networks.* 2017.

[67] GRPC. Https://grpc.io. Accessed: 2023-27-03.

[68] PROXIMAL Policy Optimization. Https://openai.com/research/openai-baselines-ppo. Accessed: 2023-28-03.

[69] BROWNE, C. B. et al. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 4, n. 1, p. 1–43, 2012.

[70] SILVER, D. et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* 2017.

[71] KIPF, T. N.; WELLING, M. *Semi-Supervised Classification with Graph Convolutional Networks.* 2017.

[72] HAMILTON, W. L.; YING, R.; LESKOVEC, J. *Inductive Representation Learning on Large Graphs.* 2018.

[73] ERDOS, P. L.; RÉNYI, A. On the evolution of random graphs. *Transactions of the American Mathematical Society*, v. 286, p. 257–257, 1984.

[74] TEST-SUITE Guide. Https://llvm.org/docs/TestSuiteGuide.html. Accessed: 2023-28-03.

[75] SCHUETZ, M. J. A. et al. Graph coloring with physics-inspired graph neural networks. *Physical Review Research*, American Physical Society (APS), v. 4, n. 4, nov 2022. Disponível em: <https://doi.org/10.1103\%2Fphysrevresearch.4.043131>.

[76] SCARSELLI, F. et al. The graph neural network model. *IEEE Transactions on Neural Networks*, v. 20, n. 1, p. 61–80, 2009.

[77] WU, F.-Y. The potts model. *Reviews of modern physics*, APS, v. 54, n. 1, p. 235, 1982.

[78] ZHOU, Y.; HAO, J.-K.; DUVAL, B. *Reinforcement learning based local search for grouping problems: A case study on graph coloring.* 2016.

[79] SHEN, Y. et al. *Enhancing Column Generation by a Machine-Learning-Based Pricing Heuristic for Graph Coloring.* 2022.

[80] KHAKHULIN, T.; SCHUTSKI, R.; OSELEDETS, I. *Graph Convolutional Policy for Solving Tree Decomposition via Reinforcement Learning Heuristics*. 2020.

[81] ZHANG, Y.; HU, X. S.; CHEN, D. Z. Efficient global register allocation for minimizing energy consumption. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 4, p. 42–53, apr 2002. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/510857.510867>.

[82] LIU, T. et al. Register allocation for simultaneous reduction of energy and peak temperature on registers. In: *2011 Design, Automation & Test in Europe*. [S.l.: s.n.], 2011. p. 1–6.

[83] GEBOTYS, C. H. Low energy memory and register allocation using network flow. In: *Proceedings of the 34th Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1997. (DAC '97), p. 435–440. ISBN 0897919203. Disponível em: <https://doi.org/10.1145/266021.266192>.