Universidade Estadual de Londrina

PEDRO ZAFFALON DA SILVA

RIGSET-UEL: UM CONJUNTO DE DADOS PARA ALOCAÇÃO DE REGISTRADORES COM APRENDIZADO DE MÁQUINA

LONDRINA
2025

PEDRO ZAFFALON DA SILVA

RIGSET-UEL: UM CONJUNTO DE DADOS PARA ALOCAÇÃO DE REGISTRADORES COM APRENDIZADO DE MÁQUINA

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

Coorientadora: Profa. Dra. Helen Cristina de Mattos Senefonte

LONDRINA
2025

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

S586r Silva, Pedro Zaffalon da.
RigSet-UEL: Um Conjunto de Dados Para Alocação de Registradores com Aprendizado de Máquina / Pedro Zaffalon da Silva. - Londrina, 2025.
105 f.

Orientador: Wesley Attrot.
Coorientador: Helen Cristina de Mattos Senefonte.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2025.
Inclui bibliografia.

1. Alocação de Registradores - Tese. 2. Aprendizado de Máquina - Tese. 3. Otimização de Compilador - Tese. 4. Conjunto de Dados - Tese. I. Attrot, Wesley. II. Senefonte, Helen Cristina de Mattos. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

CDU 519

PEDRO ZAFFALON DA SILVA

RIGSET-UEL: UM CONJUNTO DE DADOS PARA ALOCAÇÃO DE REGISTRADORES COM APRENDIZADO DE MÁQUINA

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

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

Coorientadora: Profa. Dra. Helen Cristina de Mattos Senefonte
Universidade Estadual de Londrina

Profa. Dra. Tamara Angélica Baldo
Universidade Tecnológica Federal do Paraná

Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina

Londrina, 10 de Julho de 2025.

AGRADECIMENTOS

Ao professor Wesley Attrot e à professora Helen Cristina de Mattos Senefonte, pela orientação e auxílio, que foram fundamentais para a execução deste trabalho e minha evolução acadêmica.

Aos meus pais, por todo o suporte, incentivo, ensinamentos e amor que me dedicaram a minha vida inteira.

Aos meus avós, pelo carinho e força que sempre me deram. De modo especial ao meu avô Diclovison, que buscou despertar minha curiosidade desde cedo e infelizmente hoje não está mais entre nós.

À Blenda e ao Rafael, que me acompanharam desde o início da graduação e enfrentaram esse novo desafio ao meu lado.

Aos meus irmãos e amigos, pelo apoio constante ao longo desta caminhada. Especialmente ao Marcos, por sempre acreditar em mim.

Aos professores do departamento de computação da UEL e, em especial, à professora Neyva Maria Lopes Romeiro, por todo o aprendizado proporcionado desde a graduação.

À CAPES e à Universidade Estadual de Londrina pela concessão da bolsa de estudos, que possibilitou a realização deste trabalho e do curso de mestrado.

SILVA, P. Z.. **RigSet-UEL: Um Conjunto de Dados Para Alocação de Registradores com Aprendizado de Máquina**. 2025. 105f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2025.

RESUMO

A alocação de registradores é uma etapa que impacta significativamente no desempenho de códigos gerados pelo compilador. Geralmente, sua resolução é realizada através da coloração de grafo, sendo, portanto, um problema NP-completo. Devido à sua importância, várias heurísticas foram propostas para a sua resolução. Contudo, a criação delas é um processo complexo e altamente especializado. Em um contexto atual no qual aprendizado de máquina é cada vez mais aplicado em otimizações de compiladores, sua utilização para melhorar a alocação de registradores pode se tornar uma opção interessante. Porém, devido à maior dificuldade para adaptar modelos ao problema de alocação de registradores, apenas recentemente esse tema foi mais pesquisado. Por esse motivo, há uma falta de dados de treinamento voltados para essa problemática. Neste contexto, este trabalho propõe a criação do RigSet-UEL, o primeiro conjunto de dados especializado para treinamento de modelos de aprendizado de máquina aplicados no problema de alocação de registradores. Ainda, é proposto o desenvolvimento de modelos de aprendizado de máquina para a criação de heurísticas para a alocação de registradores, demonstrando o uso do RigSet-UEL e um novo método de aplicar aprendizado de máquina para o problema.

Palavras-chave: Alocação de Registradores, Aprendizado de Máquina, Otimização de Compilador, Conjunto de Dados

SILVA, P. Z.. **RigSet-UEL: A Dataset For Register Allocation With Machine Learning**. 2025. 105p. Master's Thesis (Master in Science in Computer Science) – State University of Londrina, Londrina, 2025.

ABSTRACT

Register allocation is an important phase for compiler optimization, generally mapped to graph coloring, thus an NP-complete problem. Because of its impact on 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 have been proposed to solve compiler optimization problems. However, due to the greater difficulty in adapting models to the register allocation problem, this topic has only recently received more research attention. As a result, there is a lack of training data specifically designed for this problem. In this context, this work proposes the creation of RigSet-UEL, the first specialized dataset for training machine learning models applied to the register allocation problem. Furthermore, it also proposes the development of machine learning models to create novel heuristics to register allocation, demonstrating the use of RigSet-UEL and a novel method for applying machine learning to the problem.

Keywords: Register Allocation, Machine Learning, Compiler Optimization, Dataset

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Alocação de registradores é a etapa do compilador responsável pelo mapeamento das variáveis para o armazenamento físico do computador, na qual é decidido se cada variável será salva em um registrador ou na memória principal. É importante considerar a grande diferença entre o gasto de energia e a velocidade de acesso dos registradores e da memória principal. Por esse motivo, é necessário minimizar o armazenamento de variáveis na memória para que códigos gerados por compiladores obtenham melhor desempenho e menor consumo de energia [11, 12]. Portanto, a alocação de registradores é uma etapa importante para garantir a qualidade da geração de código.

A alocação de registradores é um problema extensamente pesquisado há décadas. Geralmente é abstraído para a coloração de grafos, um problema de otimização combinatória NP-completo [13, 14, 15], 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 próprios registradores que serão alocados, são necessárias atribuições de 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 [15, 16, 1, 17]. 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 [18]. 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 modelos de aprendizado de máquina [19, 20] em diversas áreas, abordagens baseadas em redes neurais, por exemplo, foram propostas para melhorar as otimizações realizadas pelos compiladores. Geralmente, essas abordagens são aplicadas em problemas como *phase ordering* [21, 22], *throughput prediction* [23], ou na criação de heurísticas para otimizações [24, 25, 18, 26, 27]. É importante considerar 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 à 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, abordagens baseadas em aprendizado de máquina foram aplicadas para a resolução da alocação de registradores apenas recentemente [6, 7, 28, 9, 29], as quais obtiveram resultados promissores, apesar da pequena quantidade de pesquisas envolvendo esse tema.

Embora o uso de modelos de aprendizado de máquina seja promissor, pesquisadores na área enfrentam o desafio da escassez de dados de treinamento disponíveis. Como essa área permanece relativamente inexplorada na pesquisa, conjuntos de dados dedicados a modelos relacionados à alocação de registradores são inexistentes. Consequentemente, estudos neste domínio têm recorrido a métodos alternativos para treinar seus modelos, como a geração de grafos aleatórios que não representam código do mundo real [6, 9], ou a condução de treinamento diretamente no ambiente de execução do compilador utilizando aprendizado por reforço [7]. O problema recorrente da insuficiência de conjuntos de dados especializados no contexto de alocação de registradores também tem sido observado além do escopo da pesquisa de modelos de aprendizado de máquina. Frequentemente, a escassez de casos de teste representa um desafio significativo no desenvolvimento de novos alocadores, assim como no aprendizado de técnicas de alocação de registradores.

A geração de grafos é uma alternativa simples para o problema, visto que a criação de conjuntos de dados de grafos de interferência envolveria processos como coletas de códigos-fonte, geração de seus respectivos grafos de interferência e, potencialmente, a coleta de dados de cada variável para o cálculo de custo de *spill*. Entretanto, grafos gerados aleatoriamente podem não representar as características de códigos reais, limitando o aprendizado dos modelos de aprendizado de máquina. Além disso, grafos aleatórios não apresentam os custos de *spill* das variáveis, os quais são informações importantes para o treinamento de modelos, visto que decisões de *spill* podem impactar significativamente a qualidade das alocações.

Neste contexto, é proposto neste trabalho a criação do RigSet-UEL (*Register Interference Graphs Dataset*), o primeiro conjunto de dados especializado para treinamento de modelos de aprendizado de máquina aplicados no problema de alocação de registradores. Foram coletados códigos-fonte de diversos contextos para garantir a representatividade de múltiplos tipos de códigos e foram extraídas informações das variáveis para permitir o cálculo do custo de *spill* e o uso do conjunto de dados para diferentes abstrações da alocação de registradores. Além disso, para demonstrar o uso do conjunto de dados, é proposto o uso de modelos de aprendizado de máquina para a criação de novas heurísticas de coloração para a alocação de registradores, para facilitar a inclusão dos modelos em alocadores consolidados. Este trabalho tem como objetivo facilitar e incentivar novas pesquisas na área por meio do RigSet-UEL, demonstrar o uso do conjunto de dados criado e apresentar novas formas de aplicar modelos de aprendizado de máquina na alocação de registradores. Sendo assim, espera-se que este trabalho contribua para melhorar o desempenho e o consumo de energia de códigos gerados por compiladores.

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 consiste em uma fundamentação teórica sobre as princi-

pais técnicas de aprendizado de máquina; o Capítulo 4 apresenta os principais trabalhos sobre a aplicação de aprendizado de máquina na alocação de registradores presentes na literatura; o Capítulo 5 apresenta o conjunto de dados criado e suas características; o Capítulo 6 apresenta os modelos de aprendizado de máquina para alocação de registradores desenvolvidos com o conjunto de dados; por fim, o Capítulo 7 apresenta a conclusão do trabalho.

2 TÉCNICAS TRADICIONAIS DE ALOCAÇÃO DE REGISTRADORES

Devido à 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 [30]. 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 [31, 10].

Portanto, as variáveis devem ser preferencialmente armazenadas em registradores. No entanto, a quantidade de registradores é bastante limitada. Por exemplo, processadores x86-64 apresentam apenas 16 registradores de uso geral [32] 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* [15].

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 dos códigos gerados por compiladores. Porém, a alocação de registradores é um problema NP-completo [13, 14, 15], então não existem alocadores que garantam soluções ótimas para todos os casos.

Neste contexto, diversos algoritmos foram propostos para realizar a alocação de registradores, como a coloração de grafos (Seção 2.1), o *Linear Scan* (Seção 2.2) e PBQP (Seção 2.3), 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 [33]. 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 [15].

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 representado pelo vértice. Caso não seja possível evitar *spills*, o objetivo passa a ser minimizar o seu custo [15].

2.1.1 Alocador de Chaitin

Chaitin *et al.* [15] 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:

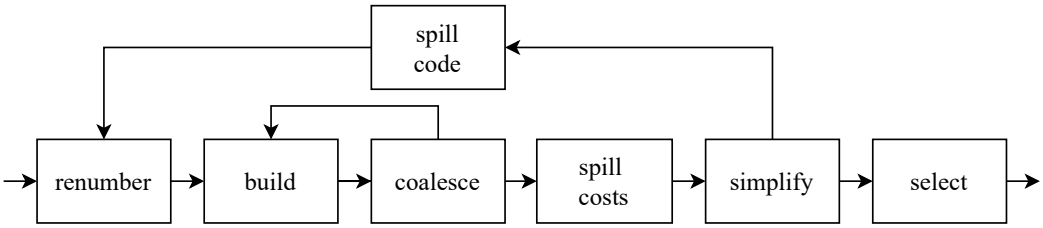


Figura 1 – Alocador estilo Chaitin. Essa figura foi adaptada do trabalho de Brigss *et al.* [1].

- ***Renumber***: Obtém 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 seja 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, um vértice é escolhido para sofrer *spill*. O vértice e suas arestas são removidos 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***: Esta etapa ocorre caso a etapa *simplify* decida realizar *spill* em alguma variável. Para que uma instrução possa utilizar o valor de uma variável que sofreu *spill* é necessário carregá-lo temporariamente a um registrador, gerando novas interferências com as demais variáveis. Portanto, 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 passos 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 ao enviar uma variável para a memória, novos tempos de vida menores são gerados. Chaitin *et al.* [15] 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 esse motivo, ao realizar *spill* de um tempo de vida, é necessário criar vários tempos de vida 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 [14]. Neste contexto, várias pesquisas foram realizadas para melhorar os seus resultados. Entre estas pesquisas, Brigss *et al.* [1] descreveram a *optimistic coloring*, que consiste na utilização de heurísticas fortes para obter resultados k-coloráveis 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 o 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.

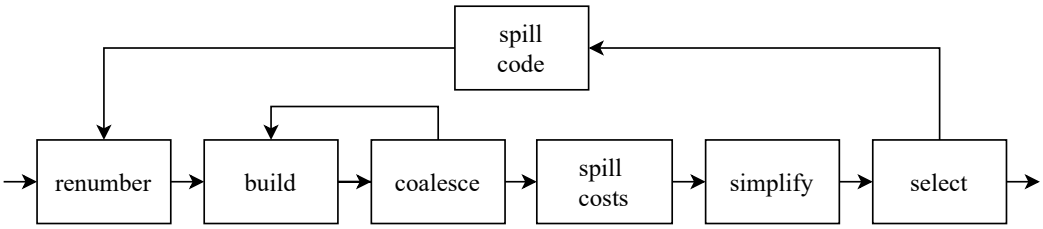


Figura 2 – Alocador estilo Briggs. Esta figura foi adaptada do trabalho de Brigss *et al.* de [1].

Estas mudanças geram duas consequências: a primeira é a possibilidade de 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.

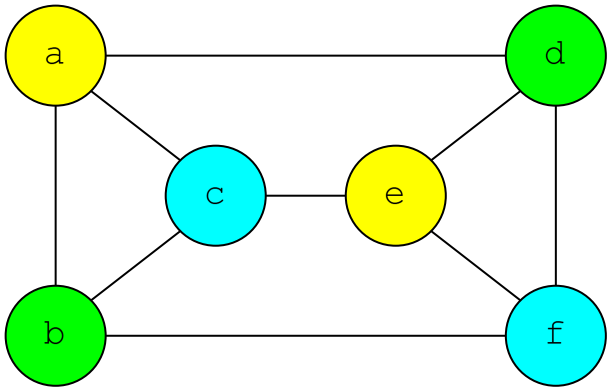


Figura 3 – Exemplo de grafo 3-colorável com vértices com 3 arestas.

Alocador de Briggs - Exemplo de Execução

O exemplo de execução é feito considerando três cores (registradores) e o bloco simples de código apresentado no Código 1:

```
1  a = 5;
2  b = a + 5;
3  c = 10 + a;
4  d = b + c;
5  e = b * d;
6  d = d + c;
7  print(e);
```

Código 1 – Bloco de código para o exemplo de execução do alocador de Briggs.

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

- A primeira definição da variável `a` está na primeira linha e o último uso na terceira;
- A primeira definição da variável `b` está na segunda linha e o último uso na quinta;
- A primeira definição da variável `c` está na terceira linha e o último uso na sexta;
- A primeira definição da variável `d` está na quarta linha e o último uso na sexta;
- A primeira definição da variável `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 ilustrado 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, as variáveis `a` e `c` não interferem entre si, apesar de serem usados na mesma instrução.

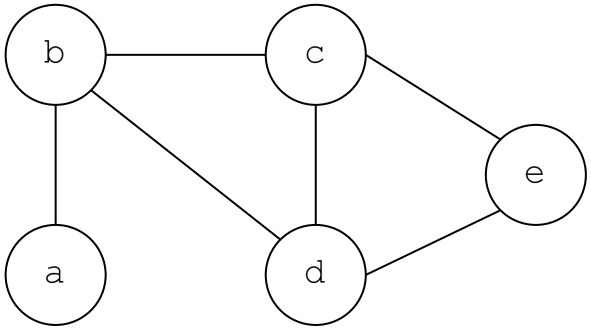
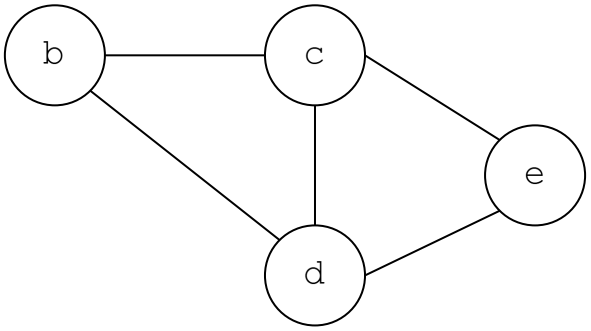


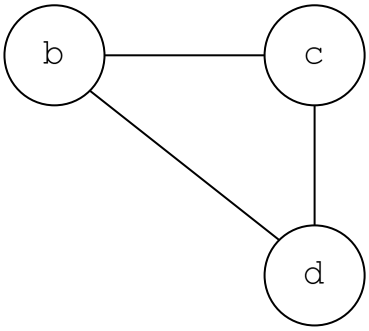
Figura 4 – Grafo de Interferência do bloco.

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

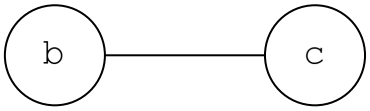
Em seguida, ocorre a fase *simplify*, que remove os tempos de vida do grafo e os adiciona em uma pilha. A remoção é realizada sempre com o vértice com menor grau no grafo. Desta forma, a variável `a` é removida inicialmente, seguida pelos demais em qualquer ordem. Essa etapa está ilustrada na Figura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



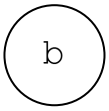
(a) Remoção de `a`.



(b) Remoção de `e`.



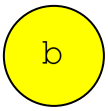
(c) Remoção de `d`.



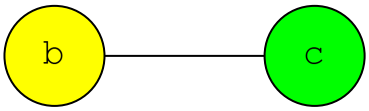
(d) Remoção de `c`, sobrando apenas `b`, que será removido em seguida.

Figura 5 – Remoção dos vértices do grafo.

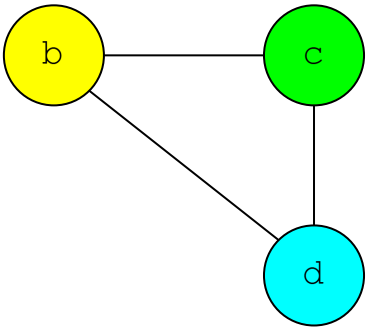
Em seguida é realizada a etapa *select*, ilustrada na Figura 6. Nesta fase os vértices são retirados da pilha, adicionados novamente no grafo e coloridos. O resultado obtido é uma coloração válida com três registradores, logo não é preciso realizar *spill* nesse exemplo.



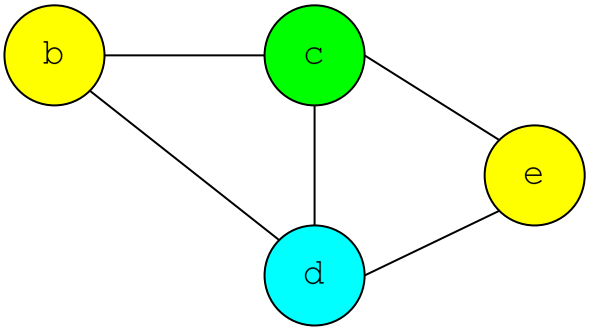
(a) Coloração e adição de b.



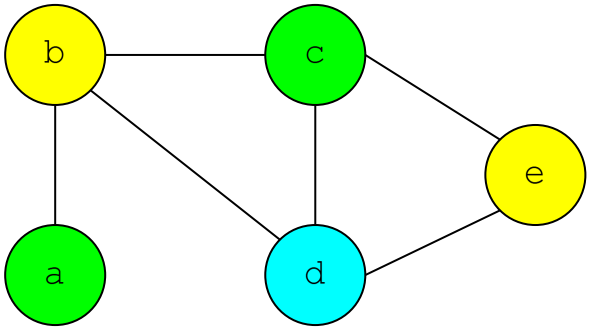
(b) Coloração e adição de c.



(c) Coloração e adição de d.



(d) Coloração e adição de b.



(e) Coloração e adição de a.

Figura 6 – Reconstrução do grafo com os vértices coloridos.

2.2 Linear Scan

Como uma alternativa para alocação de registradores baseada em coloração de grafo, foi proposto por Poletto e Sarkar [33] 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, o *Linear Scan* se tornou uma opção interessante para aplicações nas quais o tempo de compilação é uma preocupação, como compiladores dinâmicos e "*just-in-time*" [33].

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 algoritmo percorre a lista e aloca um tempo de vida em cada iteração. O número de registradores em interferência apenas muda quando algum tempo de vida termina ou começa. Sendo assim, o algoritmo pode passar de um ponto de início de um tempo de vida para o próximo.

Em cada iteração é mantida uma lista, chamada *ativos*, de tempos de vida que interferem com o tempo de vida atual e foram atribuídos a um registrador. Essa lista é mantida ordenada de acordo com pontos 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 tempo de vida atual. Devido à ordenação, a verificação pode ser encerrada ao encontrar um tempo de vida que ainda interfira com o tempo de vida atual antes do fim da lista.

O tamanho desta lista é, no máximo, o número de registradores. Caso isso aconteça em uma nova iteração e nenhum tempo de vida seja removido de *ativos*, é necessário realizar *spill*. Nesta situação, é necessário escolher uma variável para ser armazenada na memória, entre o tempo de vida da iteração atual e as variáveis presentes em *ativos*. Existem várias heurísticas para realizar esta escolha. O Algoritmo 1 [33] apresenta o algoritmo de Linear Scan.

Algoritmo 1: Linear Scan

Entrada: Lista de tempos de vida $tempos De Vida$,
número de registradores (k),
lista de registradores livres (chamado de *livres*).
Saída: Vetor associando tempo de vida e registrador alocado (chamado de *registradores*).

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

Linear Scan - Exemplo de Execução

O exemplo de execução é feito considerando três registradores e o bloco simples de código apresentado no Código 2:

```
1 a = 5;
2 b = a + 5;
3 c = 10 + a;
4 d = b + c;
5 e = b * d;
6 d = d + c;
7 print(e);
```

Código 2 – Bloco de código para o exemplo de execução do Linear Scan.

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:

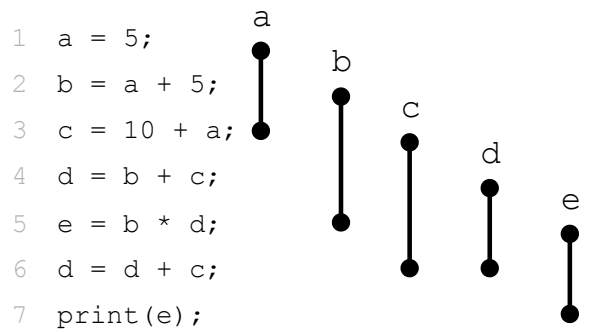


Figura 7 – Tempos de vida do exemplo de Linear Scan.

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 variável a é associado a um registrador e adicionada 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 a variável b é associado a um registrador e adicionado em *ativos*;

- Na terceira iteração a variável `a` expira. Desta forma, é removida de *ativos*, que fica na forma:

  ativos = `[b]`

  Como *ativos* continua com apenas um elemento, a variável `c` é associada a um registrador e adicionada 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 a variável `d` é associada a um registrador e adicionada em *ativos*;

- Por fim, na última iteração, ocorre a expiração da variável `b`, e *ativos* fica na forma:

  ativos = `[c,d]`

  Assim, a última variável `e` é associada a um registrador, e adicionada 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*) [34] é um caso especial do *Quadratic Assignment Problem* [35], um problema clássico de otimização combinatória que busca minimizar o custo de alocar um conjunto de n instalações a n locais, onde os custos dependem das distâncias entre os locais e dos fluxos entre as instalações. Suas principais aplicações incluem design de circuitos eletrônicos e otimização de redes de transporte [35]. O PBQP é um problema de múltiplas escolhas interdependentes e associadas a um custo. No problema, é necessário atribuir a cada variável um valor dentre um conjunto discreto de valores, buscando minimizar o custo total associado às atribuições [34]. Sua principal aplicação é na alocação de registradores, especialmente em arquiteturas irregulares [9]. Existem algoritmos com custos 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.

A Figura 9 apresenta a matriz de custo entre os vértices `a` e `b` do exemplo e a sua relação com as escolhas de alocação dos vértice. 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 a opção j de outro significa selecionar o custo da linha i e coluna j da matriz da aresta. No exemplo da Figura 9 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 finita, a seleção realizada é considerada uma solução.

O objetivo do PBQP é encontrar a solução de custo mínimo [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 às decisões de coloração, se tornando vantajosa para arquiteturas irregulares [36, 9].

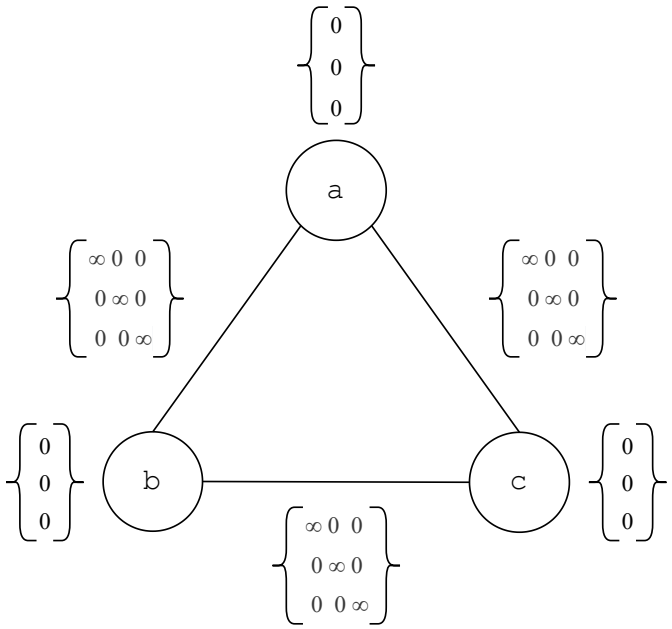
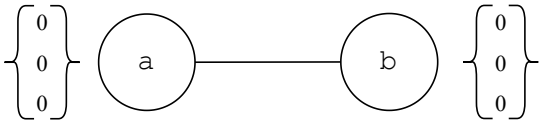


Figura 8 – Exemplo de coloração de grafos com PBQP. Essa figura foi retirada do trabalho de Buchwald *et al.* [2].



Alocação		a		
		r_1	r_2	r_3
b	r_1	∞	0	0
	r_2	0	∞	0
	r_3	0	0	∞

Figura 9 – Matriz de custo entre os vértices a e b do exemplo.

Essa abordagem foi implementada primeiramente por Scholz e Eckstein [36], 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 etapa 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 [36, 37, 38]:

- **RE**: Arestas independentes podem ser removidas após a sua matriz de custo ser decomposta em dois vetores e estes 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 10 – Exemplo de remoção RE no PBQP. Esta figura foi adaptada do trabalho de Buchwald *et al.* [2].

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

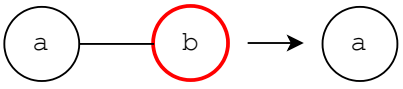


Figura 11 – Exemplo de remoção R1 no PBQP. Esta figura foi adaptada do trabalho de Buchwald *et al.* [2].

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

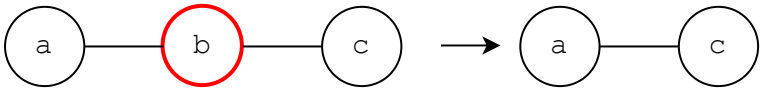


Figura 12 – Exemplo de remoção R2 no PBQP. Esta figura foi adaptada do trabalho de Buchwald *et al.* [2].

- **RN:** Para vértices com grau maior que 2 não existe remoção ótima, sendo necessário aplicar alguma heurística para realizar a redução.

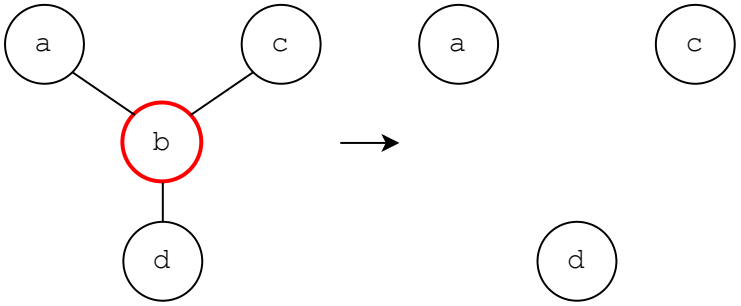


Figura 13 – Exemplo de remoção RN no PBQP. Esta figura foi adaptada do trabalho de Buchwald *et al.* [2].

As reduções são aplicadas até que o grafo não tenha interferência, tornando a solução trivial. RE, R1 e R2 são reduções ótimas, ou seja, resultam em uma solução ótima independentemente da ordem que são realizadas. 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 RN. A ordem na qual as reduções RN são realizadas pode afetar o custo final obtido, por causa disso são utilizadas heurísticas para definir a ordem na qual elas são realizadas. Neste caso, a solução ótima é perdida, mas garante tempo linear para grafos esparsos.

PBQP - Exemplo de Execução

O exemplo de execução é feito considerando três registradores e o bloco simples de código apresentado no Código 3:

O grafo PBQP desse código está ilustrado na Figura 14. A estrutura do grafo é a mesma apresentada na Figura 4 para o exemplo de coloração de grafos, mas com a adição dos vetores e matrizes de custo.

```
1 a = 5;
2 b = a + 5;
3 c = 10 + a;
4 d = b + c;
5 e = b * d;
6 d = d + c;
7 print(e);
```

Código 3 – Bloco de código para o exemplo de execução do PBQP.

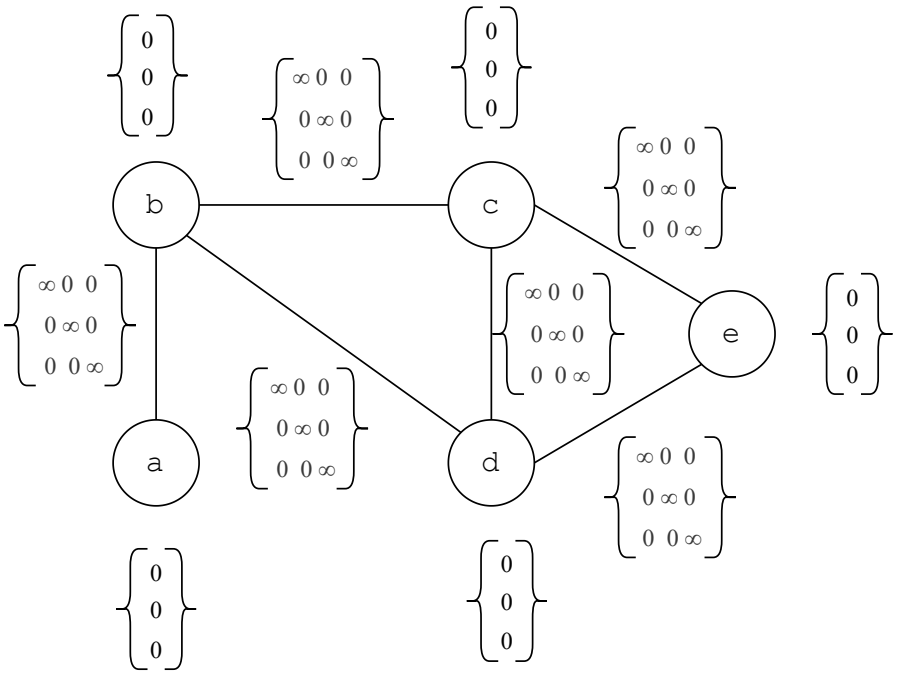


Figura 14 – Grafo PBQP do exemplo.

Para resolver a alocação de registradores por PBQP, é necessário reduzir o grafo até achar uma solução trivial. Assim, considerando que a é removido inicialmente utilizando R1, o grafo resultante é apresentado na Figura 15. Como a foi associado a um registrador, então b, seu vizinho, não pode ser associado ao mesmo registrador. Portanto, o custo deste registrador no vetor de custo de b passa a ser infinito. Ao atualizar o valor de seu vetor de custo também é necessário atualizar todas as matrizes de custos das arestas ligadas ao vértice b.

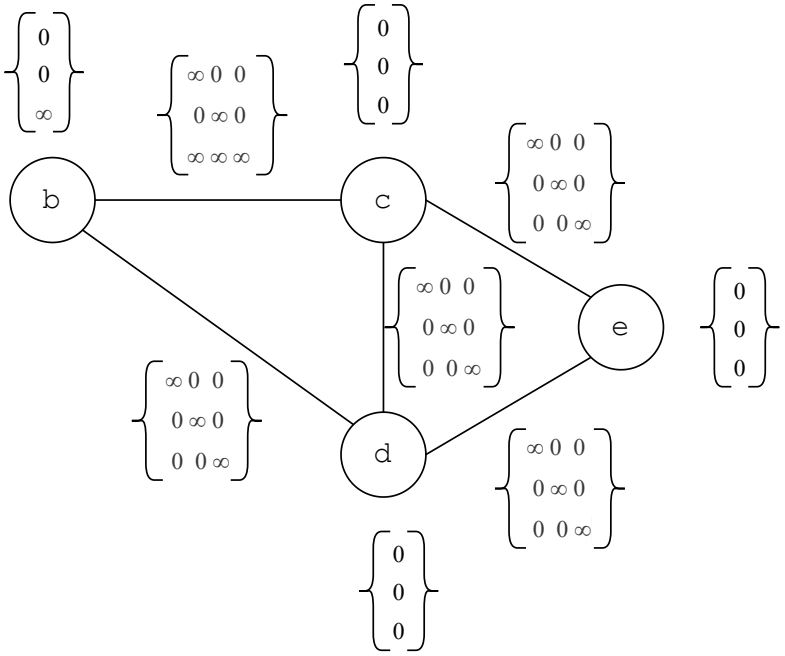


Figura 15 – Grafo PBQP do exemplo após a remoção de a.

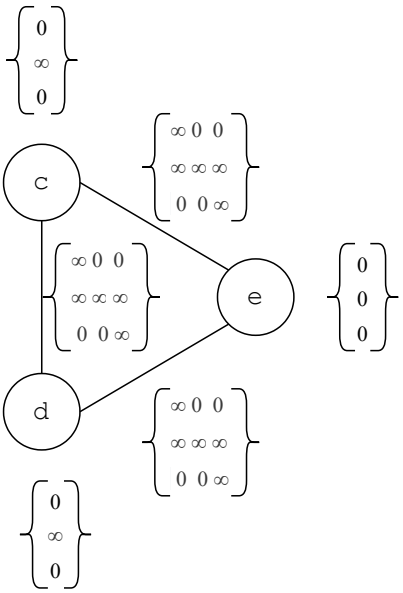


Figura 16 – Grafo PBQP do exemplo após a remoção de b.

Em seguida, b é removido utilizando R2. Após a remoção do vértice são atualizados os vetores e matrizes de custos dos vértices vizinhos. O grafo resultante é apresentado na Figura 16.

O mesmo processo ocorre ao remover c utilizando R2, assim como apresentado na Figura 17.

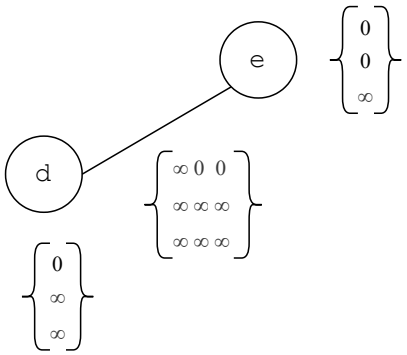


Figura 17 – Grafo PBQP do exemplo após a remoção de *c*.

A última remoção é realizada em c utilizando R2 novamente. Após essa remoção resta apenas o vértice e, tornando a solução trivial. Como existe uma opção com custo finito restante, ou seja, um registrador livre sobrando, não é necessário realizar *spill*. Desta forma, é encontrada uma solução ótima com custo zero. O grafo resultante é apresentado na Figura 18.

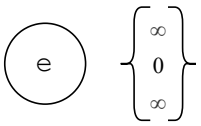


Figura 18 – Grafo PBQP do exemplo após a remoção de d.

2.3.1 Melhorias na Heurística do PBQP

Hames e Scholz [39] apresentaram uma nova heurística RN para melhorar a qualidade do código gerado. Similarmente aos algoritmos para coloração de grafos (Seção 2.1), os vértices removidos são armazenados em uma pilha. Se for necessário realizar uma remoção N, é utilizado o mesmo princípio da heurística de *optimistic coloring* de Briggs *et al.* (Seção 2.1.2), no qual os vértices são escolhidos de acordo com suas colorabilidades.

Para arquiteturas regulares, a colorabilidade depende apenas do grau do vértice. Deste modo, vértices com elevados números de interferências são removidos por último, assim como no alocador de Briggs (Seção 2.1.2). Entretanto, em arquiteturas irregulares, são necessários critérios mais robustos de colorabilidade, visto que vértices podem apresentar conjuntos diferentes de registradores disponíveis. Após todas as remoções, ocorre a fase de propagação, onde cada vértice é adicionado novamente ao grafo e associado a um registrador.

A Figura 19 apresenta o diagrama de um alocador PBQP com essa heurística. Inicialmente são realizadas remoções ótimas enquanto possíveis, enviando os vértices re-

movidos para uma pilha. Caso não existam mais remoções ótimas, os vértices restantes no grafo são removidos e adicionados na pilha conforme o critério de colorabilidade do *optimistic coloring*. Em seguida é realizada a reconstrução do grafo, similar à etapa *select* da alocação baseada em coloração de grafos. Nesta etapa os vértices são removidos da pilha e adicionados novamente no grafo e associados a um registrador livre. Após a reconstrução completa do grafo a sua alocação é obtida.

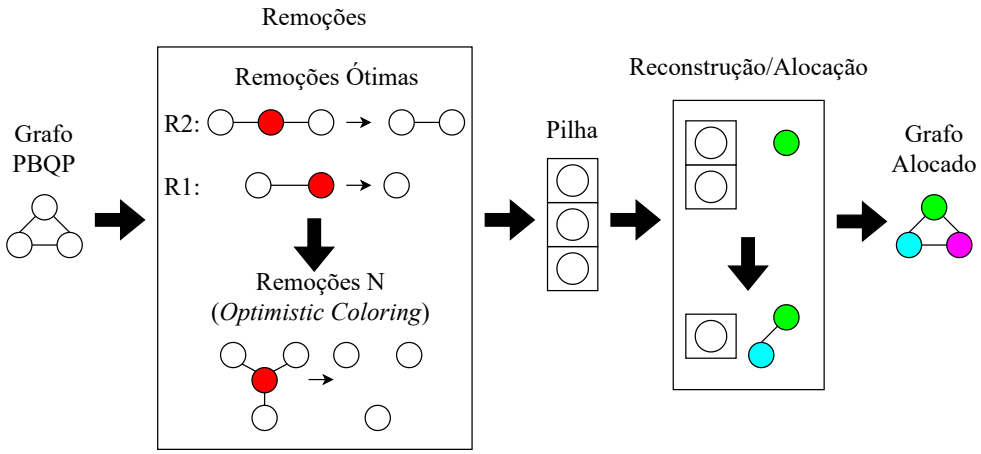


Figura 19 – Diagrama do alocador PBQP com *optimistic coloring*.

2.3.2 Branch and Bound

Hames e Scholz [39] também propuseram uma técnica *branch-and-bound* para PBQP, a qual permite encontrar soluções ótimas de forma mais eficiente que força bruta. Para obter a solução ótima, geralmente seria necessário explorar todas as possíveis sequências de remoções do grafo. Entretanto, o algoritmo *branch-and-bound* permite desconsiderar as remoções que não resultariam na solução ótima. Apesar de ser mais eficiente que força bruta, o algoritmo ainda é bastante custoso, sendo inviável sua aplicação para grafos muito grandes.

Branch-and-bound é uma técnica para resolver problemas de otimizações combinatórias discretas [39]. *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 remoções no grafo e *Bounding* em heurísticas para encontrar as melhores reduções e filtrar as soluções analisadas. Para filtrar os caminhos explorados, o algoritmo usa duas variáveis:

- ***Upper bound***: A melhor solução encontrada até o momento, ou seja, o menor custo

total de *spill* encontrado pelo algoritmo no ponto atual de sua execução;

- ***Lower bound***: O menor custo total de *spill* possível do caminho atual. É calculado sempre que uma nova remoção é explorada. Seu valor é custo de *spill* atual mais a somatória dos menores custos de alocação de cada nó que ainda não foi removido.

Caso uma remoção resulte em um *lower bound* maior que o *upper bound* atual, o caminho é desconsiderado, pois não pode resultar em um caminho melhor do que um já explorado.

A Figura 20 apresenta os estados iniciais do grafo e da árvore de decisão para um exemplo do funcionamento do algoritmo *Branch-and-bound* para alocação de registradores PBQP. Existem quatro opções de alocação, sendo as três iniciais registradores com custos de alocação variados e a última *spill*. A variável *upper bound* foi inicializada com infinito, visto que nenhuma solução foi encontrada ainda. A variável *lower bound* foi inicializada com quatro, sendo este valor a soma das opções de alocação menos custosas de todos os vértices do grafo.

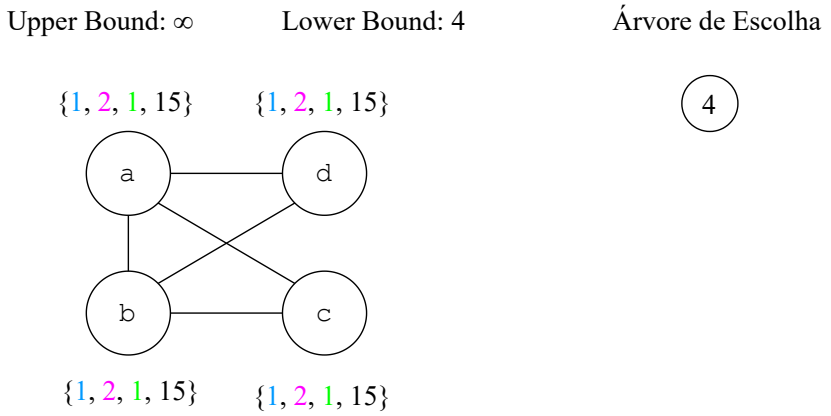


Figura 20 – Estado inicial do algoritmo *Branch-and-bound*.

A Figura 21 apresenta a primeira exploração do algoritmo, alocando o vértice *a* com a primeira opção menos custosa, o registrador representado pela cor azul com custo 1. Para facilitar a visualização, foi considerado que os vértices são coloridos, não removidos, visto que isso não afeta o funcionamento do algoritmo.

Como o valor do *upper bound* é infinito, ou seja, nenhuma solução foi encontrada ainda, nenhum caminho é descartado até que uma possível solução seja encontrada. É possível observar que os demais vetores de custos foram alterados. Como todos os outros vértices são adjacentes ao vértice *a*, estes não mais podem ser alocados com o registrador azul, desta forma, seu custo se torna infinito. Como os vetores de custo foram alterados, é necessário recalcular o *lower bound*, porém, como o registrador verde possui o mesmo custo do azul, o valor do *lower bound* se mantém igual.

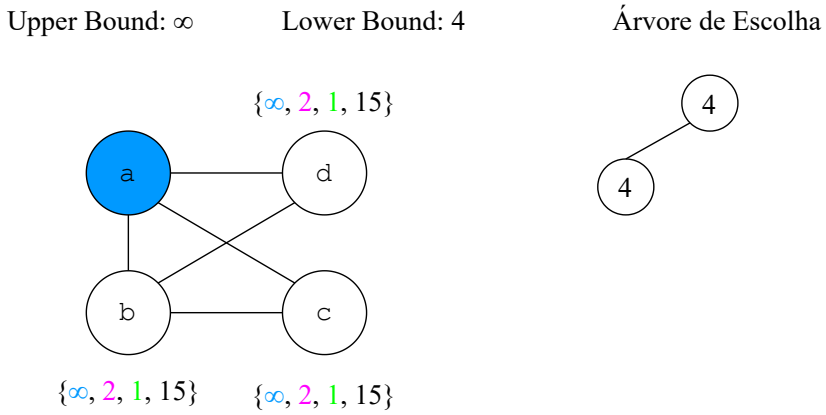


Figura 21 – Estado do algoritmo *Branch-and-bound* após explorar a como o primeiro vértice alocado.

A próxima alocação é demonstrada na Figura 22, alocando o vértice b com o registrador verde. Assim como na etapa anterior, os vetores de custo e o *lower bound* são atualizados. Como não é mais possível atribuir verde ou azul nos vértices c e d, o menor custo de ambos passa a ser dois, que é o custo do registrador magenta. Sendo assim, o *lower bound* passa a ser seis, dois dos vértices já coloridos e quatro dos menores custos dos vértices restantes.

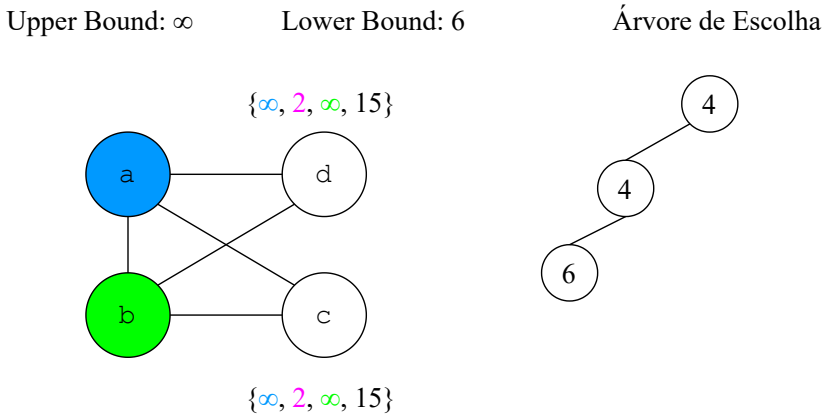


Figura 22 – Estado do algoritmo *Branch-and-bound* após explorar b como o segundo vértice alocado.

Em seguida, o vértice c é alocado com o registrador magenta, conforme a Figura 23. Como o vértice d não é adjacente ao vértice c, o seu vetor de custo permanece inalterado, assim como o valor do *lower bound*.

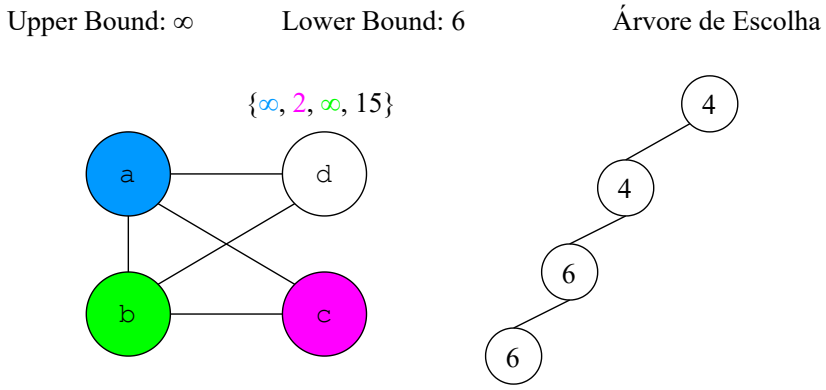


Figura 23 – Estado do algoritmo *Branch-and-bound* após explorar *c* como o terceiro vértice alocado.

Por fim, o vértice *d* é alocado com o registrador magenta, encontrando uma possível solução, conforme demonstrado na Figura 24. Como este é o último vértice, não há vetores de custo para serem alterados. Similarmente, o *lower bound* não é alterado, sendo o seu valor o custo da alocação. Entretanto, como uma solução foi encontrada, é necessário verificar se a solução encontrada é melhor que o *upper bound*. Como o valor do *upper bound* é infinito, seu valor é substituído pelo custo da alocação encontrada, ou seja, o valor do *lower bound*.

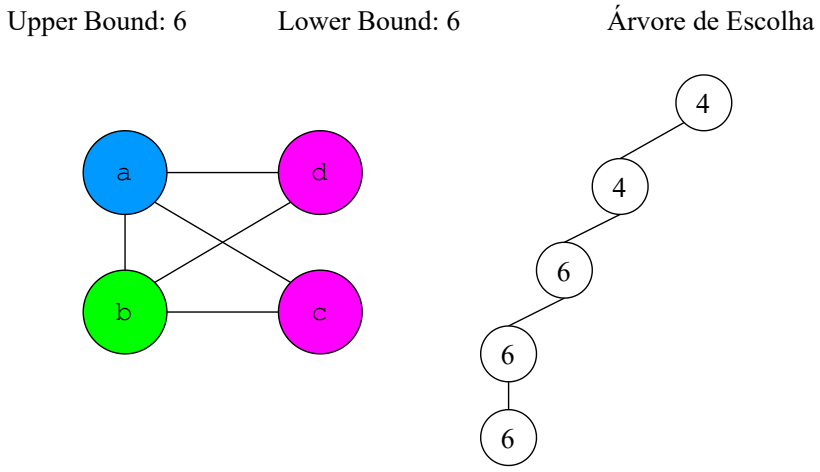


Figura 24 – Estado do algoritmo *Branch-and-bound* após explorar *d* como o último vértice alocado.

Em seguida, o algoritmo retorna para a última decisão para explorar outras alocações possíveis, para tentar encontrar outra solução melhor que a atual. Como na última decisão só existia uma possibilidade, o algoritmo retrocede novamente para explorar novas possibilidades para a penúltima alocação. O estado resultante é representado na Figura 25.

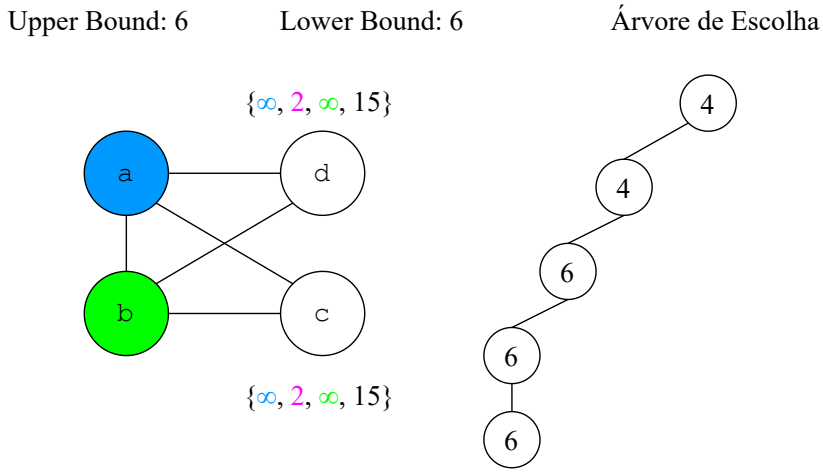


Figura 25 – Estado do algoritmo *Branch-and-bound* após retroceder para a terceira alocação para explorar outras alocações.

A Figura 26 demonstra o próximo passo do algoritmo, alocando o vértice d ao invés de c. Essa decisão de alocação resulta no *lower bound* com valor igual a seis. Como esse valor é igual ao valor do *upper bound*, esse caminho é descartado, pois é impossível resultar em uma solução com custo menor do que a melhor alocação encontrada, apenas com custo igual. Desta forma, explorar esse caminho é desnecessário.

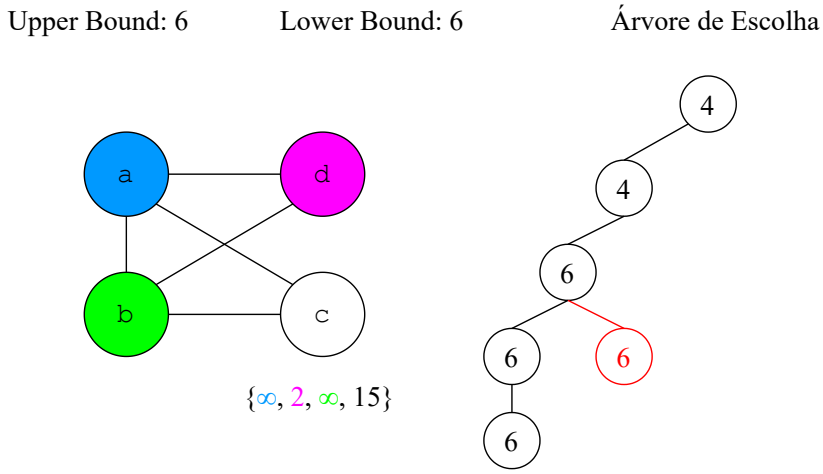


Figura 26 – Estado do algoritmo *Branch-and-bound* após explorar d como o terceiro vértice alocado.

Como todas as possíveis alocações restantes já foram exploradas, o algoritmo retrocede novamente, assim como a etapa da Figura 25. O estado do algoritmo após retroceder para a segunda alocação é apresentado na Figura 27.

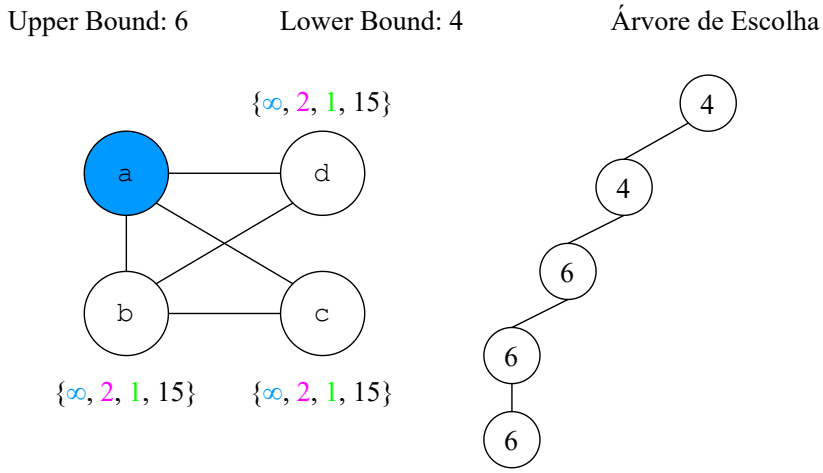


Figura 27 – Estado do algoritmo *Branch-and-bound* após retroceder para a segunda alocação para explorar outras alocações.

Assim como a Figura 26, a Figura 28 demonstra outra escolha de alocação sendo explorada. Desta vez é alocado o vértice c ao invés de b. Como o *lower bound* resultante é igual ao valor do *upper bound* este caminho também é descartado.

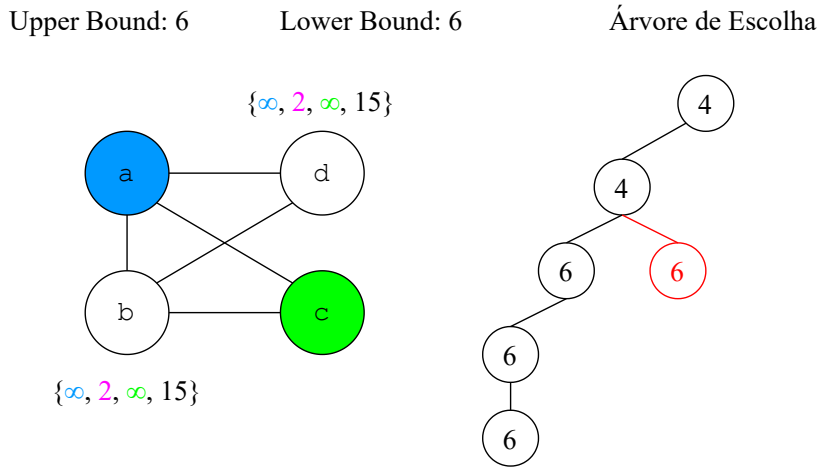


Figura 28 – Estado do algoritmo *Branch-and-bound* após explorar c como o segundo vértice alocado.

Em seguida é explorada a próxima possibilidade de alocação, na qual o vértice d é alocado com o registrador verde. Como o vértice d não é adjacente ao vértice c, este mantém a possibilidade de ser colorido com verde. Desta forma, o valor do *lower bound* é cinco. Como este valor é menor que o *upper bound*, este caminho é explorado. O estado resultante desta exploração é representado pela Figura 29.

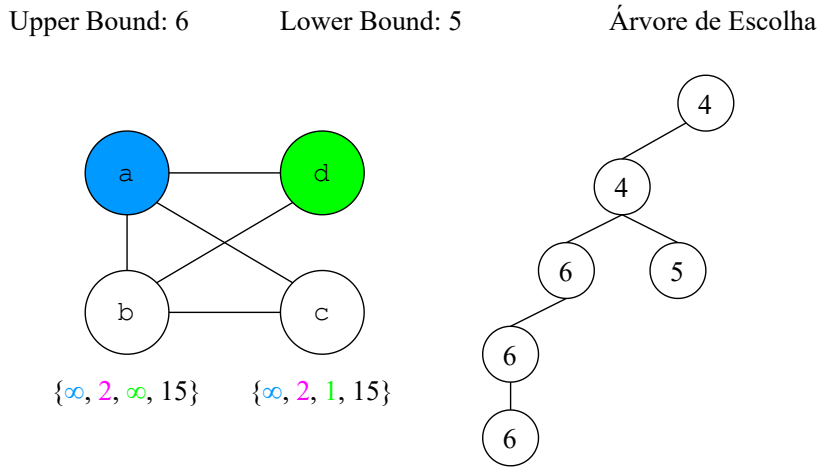


Figura 29 – Estado do algoritmo *Branch-and-bound* após explorar d como o segundo vértice alocado.

A Figura 30 apresenta a próxima exploração deste caminho, alocando o vértice b com magenta. Como o valor do *lower bound* se mantém como cinco, esse caminho continua sendo explorado.

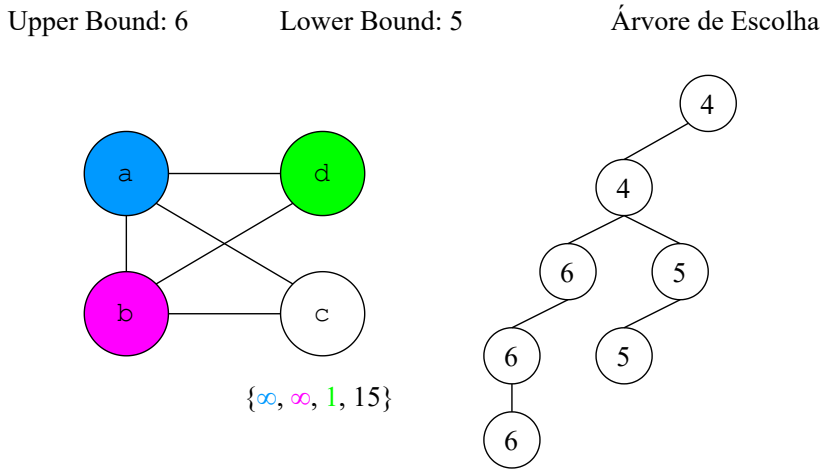


Figura 30 – Estado do algoritmo *Branch-and-bound* após explorar b como o terceiro vértice alocado.

Por fim, a Figura 31 apresenta a última exploração deste caminho, colorindo o vértice c como verde. O custo da solução encontrada é cinco, melhor que o custo salvo no *upper bound*, assim a melhor solução encontrada é atualizada.

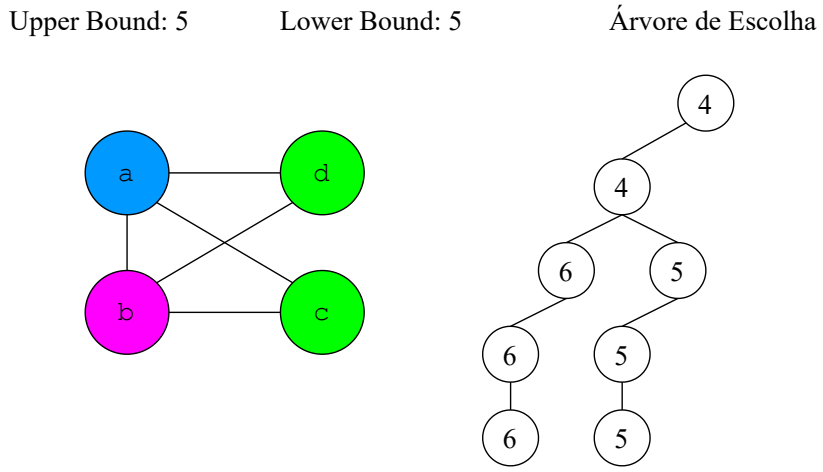


Figura 31 – Estado do algoritmo *Branch-and-bound* após explorar *c* como o último vértice alocado.

Apesar do exemplo se encerrar, o algoritmo continuaria verificando os demais caminhos ainda não explorados para tentar encontrar uma solução melhor que a atual. Além disso, este exemplo ignora outro fator importante da alocação PBQP, as remoções R1 e R2. Se uma decisão de alocação já for ótima, não é necessário explorar outras possibilidades. Sendo assim, antes de cada alocação explorada no algoritmo, seriam realizadas remoções ótimas enquanto elas existirem. A Figura 32 apresenta o diagrama do algoritmo *Branch-and-bound* para alocação PBQP.

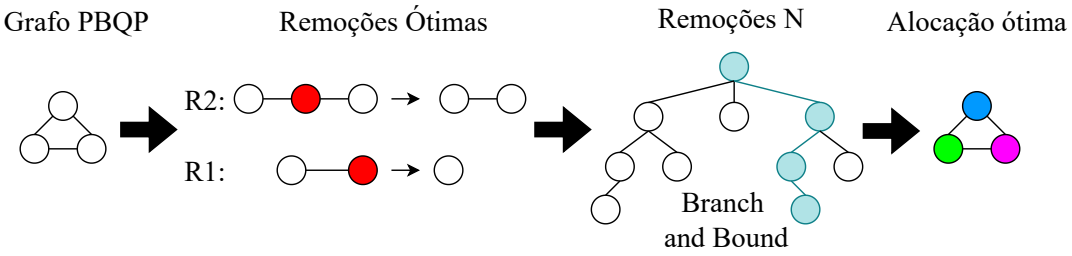


Figura 32 – Diagrama do algoritmo *Branch-and-bound* para alocação PBQP

2.4 Decisões de Spill

Em muitos caso é impossível evitar a realização de *spill*, mesmo considerando alocações ótimas. Desta forma, é importante criar heurísticas eficientes para minimizar ao máximo possível o custo da realização de *spill*.

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 resulta em grafos de interferência diferentes. Boas heurísticas de remoções podem evitar a necessidade de realizar *spill* novamente.

Inicialmente, o alocador de Chaitin [15] 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.* [40] 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 estruturas de repetição custam dez vezes mais que instruções fora. Assim, o peso de cada instrução de definição ou de uso é a sua profundidade em laços. Uma utilização com profundidade dois, ou seja, dentro de dois laços, apresenta peso cem.

Além disso, o custo é dividido 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, considerando v uma variável e i o conjunto de instruções que usam ou definem v, o cálculo do custo de *spill* ($custo$) e a heurística de alocação (h) de v são descritos nas seguintes equações:

$$custo(v) = \sum_{I \in \{\text{definições e usos de } v\}} 10^{profundidade(I)} \tag{2.1}$$

$$h(v) = \frac{custo(v)}{grau(v)} \tag{2.2}$$

Chaitin *et al.* [40] 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. 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, tornando

desnecessário adicionar a instrução *load* na segunda utilização [41], conforme ilustrado na Figura 33.

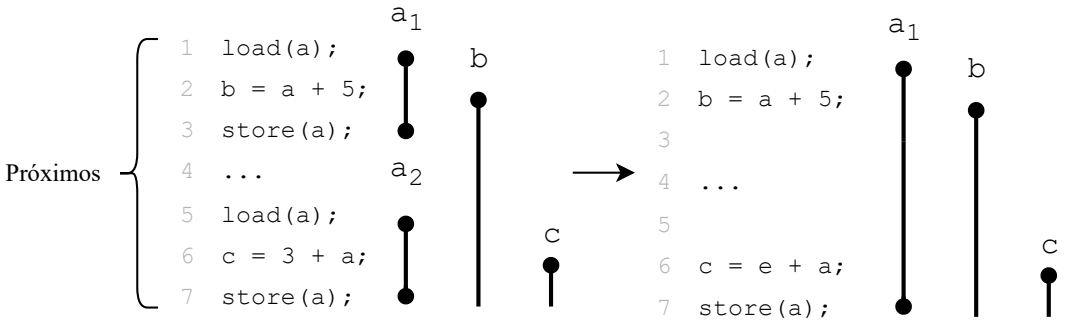


Figura 33 – Exemplo de minimização de custo de *spill* baseada em proximidade.

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.

2.5 Comparação entre as Abordagens de Alocação

Nas seções anteriores foram apresentadas as três abordagens de alocação de registradores globais mais utilizadas [11]. Nesta seção, 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 CPU 2006 para comparar diferentes técnicas de alocação de registradores. As comparações são realizadas em duas arquiteturas: x86-64 e ARM Cortex-A8. Os testes foram realizados utilizando o *framework* do LLVM [42]. São avaliados os seguintes alocadores:

- **Briggs**: Alocador de registradores por coloração de grafos utilizando *optimistic 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* [43] 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**

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 [31].

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**

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 [33]. Sua velocidade é interessante para aplicações nas 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 resultados piores. Apresentou números totais de código de *spill* inseridos melhores no SPEC CFP 2006 por uma margem baixa. Porém, obteve resultados consideravelmente 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 à sua flexibilidade [9].

3 APRENDIZADO DE MÁQUINA

Este capítulo apresenta uma fundamentação teórica sobre as técnicas de aprendizado de máquina, redes neurais e aprendizado profundo, além de suas formas de treinamento. Muitas vezes esses termos são utilizados como sinônimos, porém cada um se refere a uma subárea diferente dentro da área da inteligência artificial (IA), conforme ilustrado na Figura 34. Assim, aprendizado de máquina é um ramo da inteligência artificial, redes neurais são um dos tipos de modelos de aprendizado de máquina e aprendizado profundo uma classe de algoritmos que se baseiam em redes neurais [44, 45].

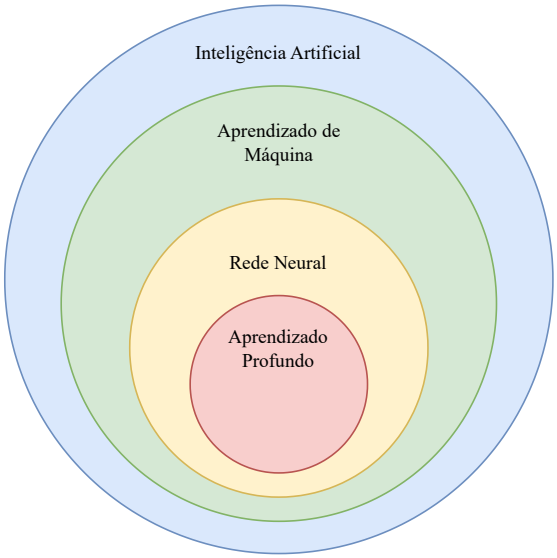


Figura 34 – Inteligência artificial e suas subáreas.

Aprendizado de máquina, especificamente, é um ramo da inteligência artificial que tem como objetivo o desenvolvimento de algoritmos que possam emular a inteligência humana, se adaptando, tomando decisões e reconhecendo padrões com base em experiências acumuladas através da solução bem-sucedida de problemas anteriores [19, 20].

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 à tomada de decisões, de forma a melhorar os resultados do algoritmo [46].

Aprendizado de máquina pode ser utilizado para diversas tarefas, entre as mais comuns estão [47]:

- **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 [48];
- **Regressão:** Uma tarefa na qual é necessário prever um valor contínuo, como preço, idade, salário, entre outros [49];
- ***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 [50];
- ***Clustering***: Consiste em encontrar padrões de similaridade dentre os dados de entrada para dividi-los em grupos (*clusters*), buscando aumentar a similaridade entre os elementos de um mesmo grupo (*cluster*) [51].

3.1 Tipos de Aprendizado

Existem várias formas de treinar modelos de aprendizado de máquina, e dependendo do problema e dos dados disponíveis, diferentes tipos de cenários de aprendizado podem ser utilizados. Os tipos de aprendizado mais comuns são [47]:

- **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 aprendizado de máquina recebe exclusivamente dados não rotulados. Geralmente é utilizado para agrupamento e identificação de padrões em dados;
- **Aprendizado por reforço**: Neste cenário o modelo aprende através de reforços (recompensas ou punições), de acordo com suas ações. Este cenário é explicado detalhadamente na próxima subseção.

3.1.1 Aprendizado por Reforço

Como na alocação de registradores é inviável rotular todos os resultados possíveis, as abordagens baseadas em aprendizado de máquina apresentadas no próximo capítulo utilizam esta forma de aprendizado. O aprendizado por reforço é composto pelos seguintes elementos [52]:

- **Agente**: O modelo aprendizado de máquina que deve tomar decisões e aprender com as experiências adquiridas;
- **Ambiente**: A representação do problema a ser resolvido, com a qual o agente interage;
- **Estado**: Uma representação do ambiente em um certo momento. O estado encapsula toda a informação relevante que o agente precisa considerar ao decidir sua próxima ação;
- **Ação**: As decisões ou movimentos que o agente pode tomar. Cada ação pode resultar em diferentes estados do ambiente;
- **Reforço**: Um valor numérico fornecido ao agente após uma ação. A recompensa pode ser positiva (recompensa) ou negativa (punição), dependendo do impacto da ação no objetivo final. A recompensa avalia a tomada de decisão do agente, permitindo o seu aprendizado.

Nesse cenário de aprendizado, 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 [52]. Para coletar informação e aprender, o modelo interage ou afeta o ambiente, recebendo um reforço para cada ação. O objetivo do modelo é maximizar a recompensa durante as interações com o ambiente [47]. Este cenário de aprendizado está ilustrado na Figura 35.

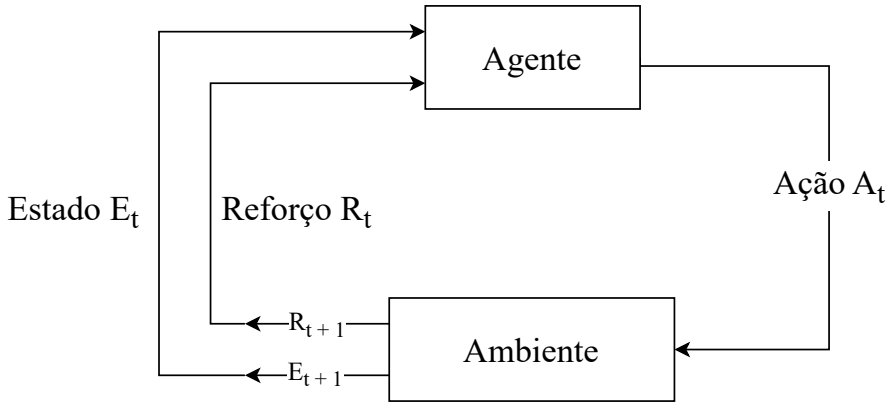


Figura 35 – Diagrama do aprendizado por reforço. Esta figura foi adaptada do trabalho de Sutton *et al.* [3].

No aprendizado por reforço 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*) [47].

Além disso, aplicações de aprendizado por reforço podem ser modeladas com mais de um agente, sendo chamado de aprendizado por reforço multi-agente [53, 54]. 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 aprendizado por reforço multi-agente pode apresentar variações em relação ao ambiente. Os agentes podem agir sequencialmente ou formar uma hierarquia, resultando no aprendizado por reforço multi-agente hierárquico [55]. Nessa modelagem, os agentes no topo da hierarquia determinam como os agentes em níveis menores devem agir [54, 7].

3.2 Redes Neurais

As Redes Neurais (Neural Networks) são um tipo de algoritmo de aprendizado de máquina 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 [56].

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 [57]. Em 1975, Kunihiko Fukushima apresentou o conceito de rede neural multicamadas [58], que continua sendo utilizado até hoje. A Figura 36 apresenta um exemplo de rede neural multicamadas.

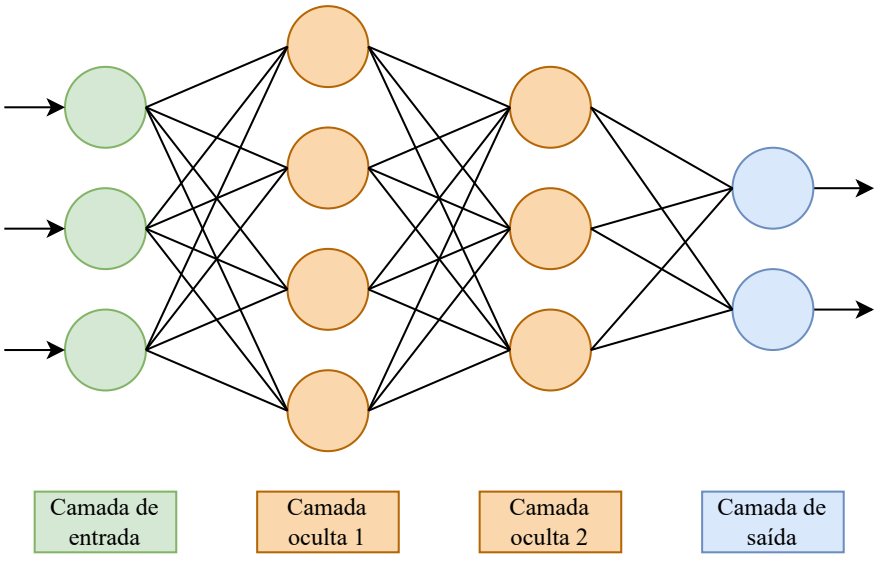


Figura 36 – Exemplo de uma rede neural multicamadas.

Conforme apresentado na Figura 36, as camadas podem ser classificadas em três grupos:

- **Camada de entrada**: Recebe os atributos da entrada, 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 ou previsão da rede neural.

Os atributos 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 viés 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 37.

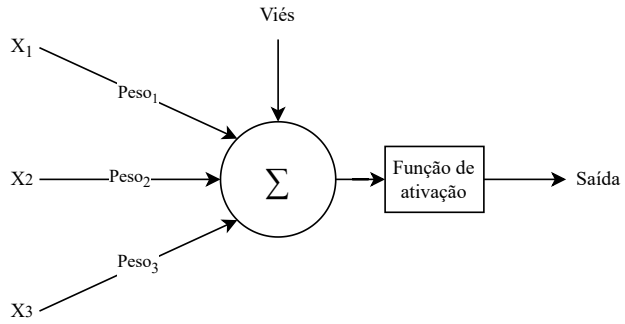


Figura 37 – Estrutura de um neurônio. Esta figura foi adaptada do trabalho de Parmezan *et al.* [4].

O processo de aprendizado em uma rede neural envolve o ajuste dos pesos e vieses de cada neurônio 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 *backpropagation* [59].

3.2.1 Aprendizado Profundo

Aprendizado profundo se baseia no uso de redes neurais, caracterizado pelo uso de arquiteturas com muitas camadas ocultas, possibilitando a modelagem de padrões complexos e não lineares [60]. Redes neurais com três ou mais camadas podem ser consideradas

algoritmos de aprendizado profundo [44]. Devido ao maior número de camadas, algoritmos de aprendizado profundo possuem maior capacidade de processamento, resultando em maior capacidade de abstração e generalização dos dados. Por este motivo, geralmente, esses algoritmos são utilizados para problemas mais complexos [61, 62].

Ao contrário de outras formas de aprendizado de máquina, que podem exigir que os dados sejam estruturados e apresentados de maneira específica, o aprendizado profundo é capaz de aprender a partir de dados brutos, como imagens, áudio e texto, sem a necessidade de pré-processamento de dados. [44]. Entre as arquiteturas mais conhecidas utilizadas em aprendizado profundo estão as CNNs (Redes Neurais Convolucionais) [63], RNNs (Redes Neurais Recorrentes) [64], LSTMs (Long Short-Term Memory) [65] e Transformers [66].

3.3 Graph Neural Network

Uma *Graph Neural Network* (GNN) é um tipo de rede neural voltado para aplicações em grafos. Conforme visto no Capítulo 2, geralmente a alocação de registradores é representada por meio de grafos. Entretanto, no contexto de redes neurais, existem diversos desafios ao se trabalhar com grafos. Modelos de aprendizado de máquina geralmente utilizam matrizes retangulares ou vetores como entrada. Portanto, a representação de grafos de maneira compatível com redes neurais não é imediatamente intuitiva.

Dados relacionados com os vértices e arestas, embora possam apresentar um número variável de exemplos, são simples de representar e podem ser processados sem técnicas especiais. No entanto, representar a conectividade de um grafo para uma rede neural é uma tarefa complexa [5].

A opção mais intuitiva provavelmente seria utilizar matrizes de adjacência, pois podem ser facilmente adequadas como entradas de um modelo de rede neural. Entretanto, algumas características de matrizes de adjacência as tornam inviáveis para sua utilização em conjunto com aprendizado de máquina. Grafos grandes geralmente resultam em matrizes muito esparsas, ocupando muito espaço desnecessário na entrada da rede neural. Um único grafo pode ser representado por diversas matrizes. Não existe garantia de que uma rede neural vai gerar o mesmo resultado para matrizes equivalentes [5].

Sendo assim, a melhor forma de representar grafos é por meio de listas. Esta representação consiste em listas contendo os atributos associados a cada vértice e aresta, além da lista de adjacência, que informa a conectividade do grafo. A Figura 38 contém um exemplo dessa representação, com uma lista indicando os vértices e seus respectivos atributos, outra contendo os atributos das arestas e, por fim, uma lista indicando as conexões formadas por cada aresta. É possível observar que, neste exemplo, cada vértice

e aresta estão associados a um único atributo, porém elas podem estar associadas a múltiplos atributos [5, 67].

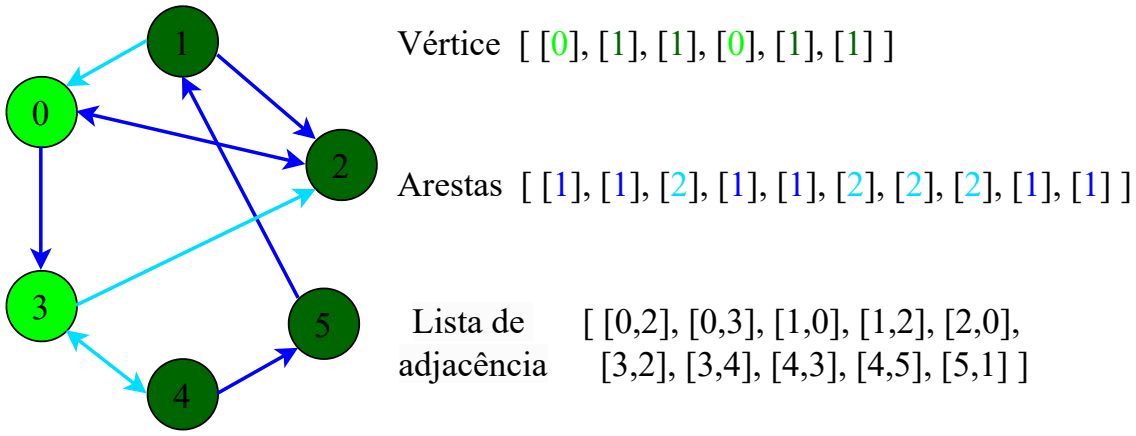


Figura 38 – Representação da entrada de uma GNN. Esta figura foi adaptada do trabalho de Sanchez-Lengeling *et al.* [5].

Geralmente, a GNN realiza transformações nos atributos dos grafos sem alterar sua conectividade. A GNN aplica transformações de forma independente a cada elemento do grafo. Cada vetor de características associado a um vértice ou aresta é processado pela rede neural, que retorna um novo conjunto de valores para cada elemento. Como a GNN não altera as conectividades do grafo, após a saída de uma camada GNN é possível utilizar a mesma lista de adjacência para representar o grafo, mas com as listas de atributos atualizadas [67]. Um exemplo de uma GNN transformando os atributos dos vértices de um grafo é apresentado na Figura 39. No exemplo, cada vértice possui inicialmente um conjunto de três atributos, os quais são transformados em um novo conjunto de dois atributos cada.

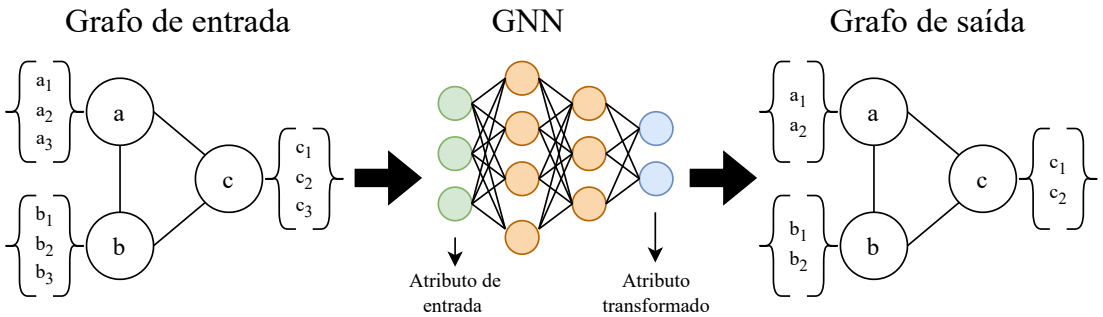


Figura 39 – Exemplo de uma GNN transformando os atributos dos vértices de um grafo.

As listas de adjacência permitem representar as conectividades de grafos, porém, apenas isso não é o suficiente para que a rede neural consiga realizar tarefas complexas que dependem da conectividade do grafo. Conforme ilustrado na Figura 39, cada elemento

do grafo sofre transformações de acordo com seus próprios atributos, independente de elementos próximos no grafo. Para resolver esse problema, as GNNs realizam *message passing*, onde vértices podem agregar informações presentes em vértices adjacentes [5].

Além disso, é possível realizar *pooling*, uma transformação que altera a conectividade do grafo. Essa operação consiste em mesclar elementos próximos através da agregação de seus atributos. *Pooling* permite a redução do tamanho do grafo, facilitando a sua classificação global [68]. A Figura 40 apresenta um exemplo de *pooling*.

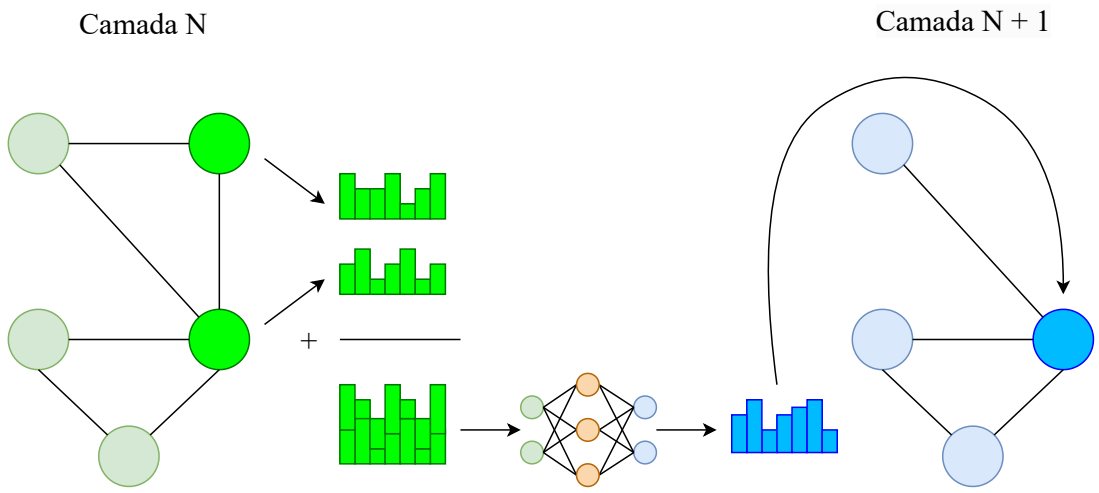


Figura 40 – Exemplo de *pooling*. Esta figura foi adaptada do trabalho de Sanchez-Lengeling *et al.* [5].

4 ALOCAÇÃO DE REGISTRADORES COM APRENDIZADO DE MÁQUINA NA LITERATURA

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 abordagens de alocação de registradores baseadas em aprendizado de máquina.

Neste contexto, este capítulo apresenta as principais técnicas propostas na literatura para resolver o problema de alocação de registradores usando aprendizado de máquina. Além disso, são apresentados trabalhos que, apesar de não abordarem o tema diretamente, apresentam técnicas que podem auxiliar no desenvolvimento de alocadores utilizando aprendizado de máquina.

A partir das abordagens que serão apresentadas nesse capítulo, será possível observar o problema de falta de conjuntos de dados especializados, além de verificar as necessidades que um conjunto de dados voltado para modelos aplicados na alocação de registradores deve atender.

4.1 Algoritmo de Coloração de Grafo Aproximada Baseada em Aprendizado Profundo

Das *et al.* [6] desenvolveu um modelo de aprendizado profundo para ser utilizado no contexto da alocação de registradores. O modelo associa um registrador a cada vértice de um grafo de interferência, buscando utilizar o menor número de registradores possível. É importante observar que, apesar de atuar no contexto da alocação de registradores, o modelo resolve um problema diferente, visto que na alocação de registradores o objetivo é alocar da melhor maneira possível um número finito de registradores.

O modelo proposto utiliza 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 [69, 64].

Especificamente, foi utilizado o *Long short-term memory* (LSTM) [65], uma variante de RNN. O LSTM é uma arquitetura de *Recurrent Neural Network* que lembra valores em intervalos arbitrários. A Figura 41 demonstra como resultados anteriores podem afetar o processamento das próximas entradas em uma LSTM, onde A é uma célula LSTM, E_n é a entrada n e S_n é a saída n.

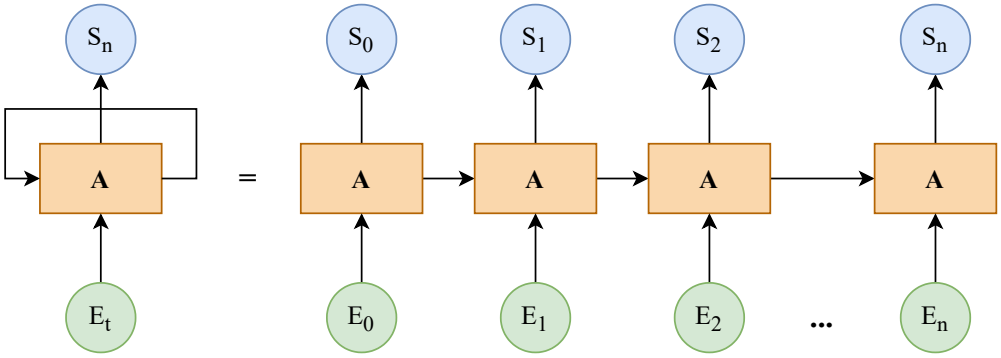


Figura 41 – Exemplo do funcionamento de uma RNN/LSTM. Esta figura foi adaptada do trabalho de Das *et al.* [6].

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 inteira do grafo 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 melhorar a eficiência das previsões, foi utilizado aprendizado profundo LSTM com três camadas. A Figura 42 ilustra essa modelagem.

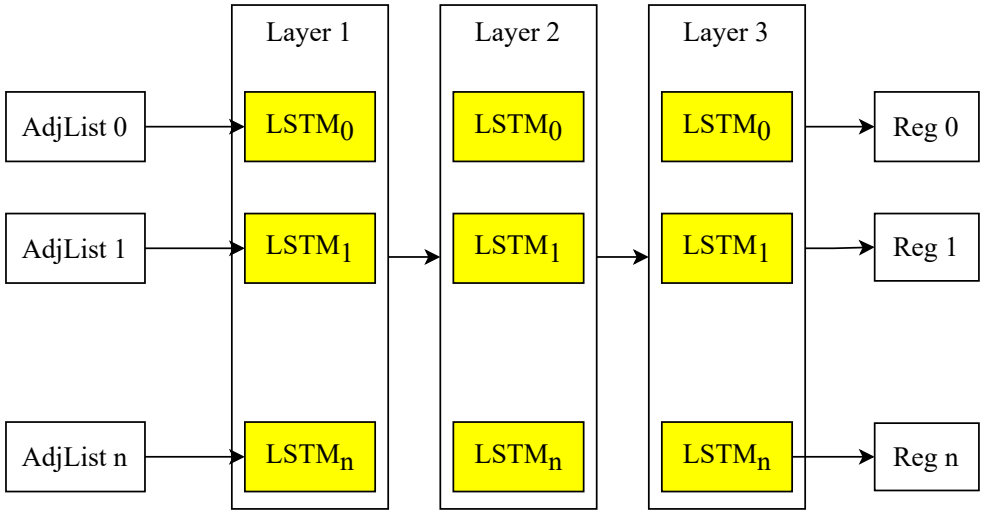


Figura 42 – Ilustração do modelo de Das *et al.* [6].

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

Foram utilizados grafos aleatórios para o treinamento do modelo. Os grafos foram gerados pelo *very nauty* [70], uma biblioteca de C de algoritmos de grafos, especialmente

voltada para geração rápida de grafos aleatórios. O treinamento foi realizado com aprendizado supervisionado, no qual a saída é comparada com uma possível solução ótima.

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 a opção de alocação *Greedy* do LLVM, que utiliza *Linear Scan* com *live range splitting* [43] agressivo. A comparação é realizada de acordo com o número de registradores necessários para obter uma coloração válida sem necessidade de *spill* com os algoritmos.

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 os demais *benchmarks*: `505.mcf_r`, `557.xz_r`, `541.leela_r` e `502.gcc_r`, os resultados foram 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 aprendizado de máquina para criar novas heurísticas para o problema de alocação de registradores, sendo necessário realizar novas pesquisas sobre o tema para criar modelos mais poderosos.

4.2 Alocação de Registradores com Aprendizado por Reforço

VenkataKeerthy *et al.* [7] criaram uma aplicação baseada em aprendizado por reforço que executa toda a alocação de registradores, além de implementarem o LLVM-gRPC [7], que permite fácil integração de aprendizado de máquina 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, estocásticos e sequenciais [71]. Além disso, foi utilizado aprendizado por reforço multi-agente hierárquico [55], dividindo a alocação de registradores 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***: Agente de nível baixo que aprende a escolher uma 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 é 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 que decide se a variável escolhida é alocada para um registrador ou sofre o processo de *live range splitting* [43]. 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 ocorrer;
- ***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 [72]. Os grafos são utilizados como entrada para um modelo de *Gated Graph Neural Network* (GGNN) [73], 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 [74] e a ferramentas do LLVM, aproveitando a estrutura modulada do compilador. A Figura 43 apresenta um diagrama do funcionamento do alocador e sua integração com o LLVM.

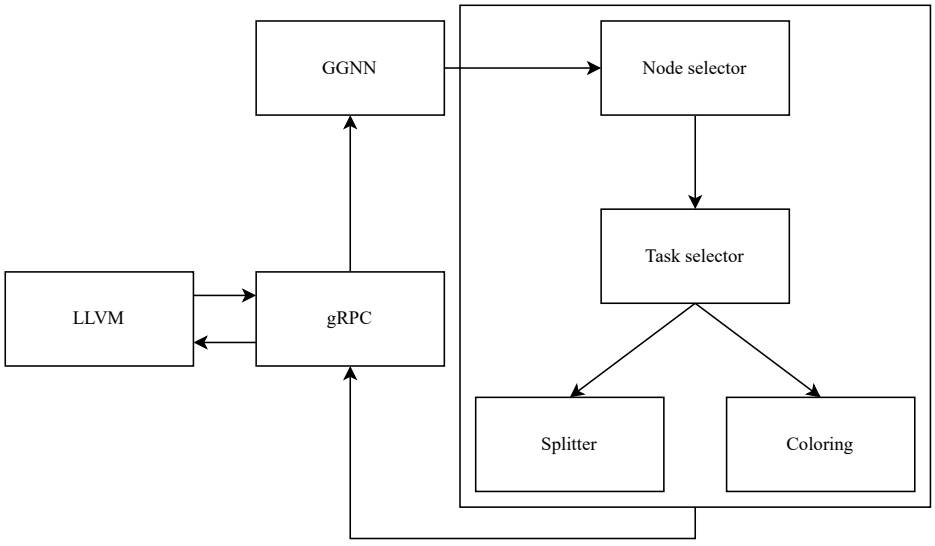


Figura 43 – Ilustração simplificada do funcionamento do alocador de VenkataKeerthy *et al.* [7].

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 aprendizado por reforço são treinados utilizando *Proximal Policy Optimization* (PPO) [75]. 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 obtiveram 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.

4.3 Alocação de Registradores PBQP com Aprendizado por Reforço Profundo

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 proposta uma abordagem de aprendizado por reforço para resolver a alocação de registradores baseada em PBQP.

O alocador utiliza Monte Carlo *tree search* (MCTS) [76], uma técnica de busca baseada em heurísticas e probabilidades, combinando elementos da implementação clássica do *tree search* com princípios de aprendizado por reforço. 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

[76]. 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;
- ***Expansion***: 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.

Estas etapas são repetidas até que a solução seja encontrada. A Figura 44 ilustra o funcionamento do Monte Carlo *tree search*.

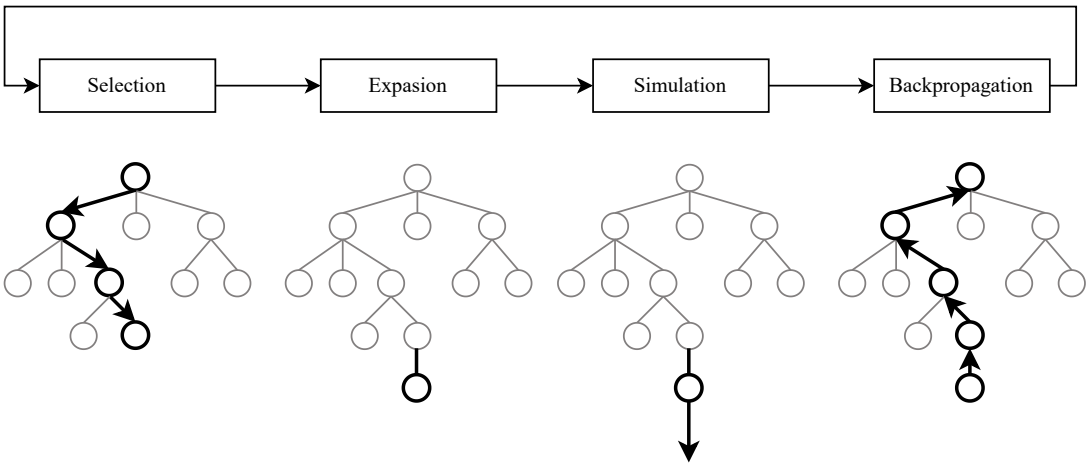


Figura 44 – Ilustração do funcionamento do Monte Carlo *tree search*. Esta figura foi adaptada do trabalho de Chaslot *et al.* [8].

Além disso, também foi utilizado o modelo de aprendizado por reforço profundo *AlphaZero*, que foi desenvolvido pela empresa de pesquisa em inteligência artificial DeepMind para dominar os jogos de xadrez, shogi e go [77]. 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 *AlphaZero* para obter os resultados das simulações.

Para utilizar o modelo *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 [78, 79]. 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 45 apresenta um diagrama do funcionamento do modelo Kim *et al.*

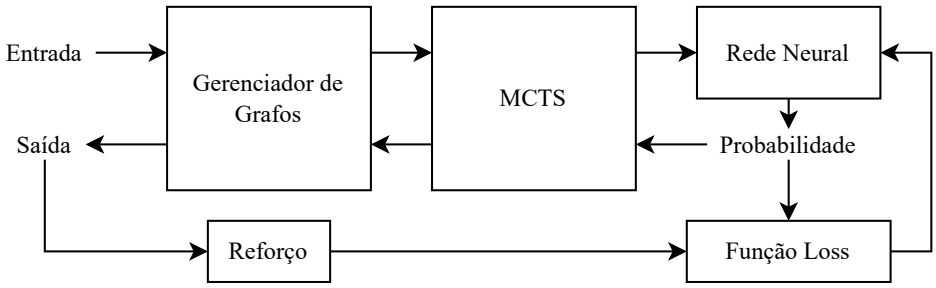


Figura 45 – Ilustração simplificada do modelo de Kim *et al.* [9].

O modelo foi treinado utilizando grafos aleatórios gerados pelo modelo de Erdos-Rényi [80]. Para verificar o desempenho do modelo, foram utilizados 24 *benchmarks* presentes no LLVM *test-suite* [81]. O alocador de Kim *et al.* foi comparado com as opções *greedy*, PBQP e *fast* do LLVM. A opção *fast* é baseada em alocação de registradores local [82] e foi utilizada 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*.

4.4 Outros Trabalhos

Apesar de não abordarem diretamente a alocação de registradores, outros trabalhos apresentam aplicações de aprendizado de máquina 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 aprendizado de máquina na alocação de registradores.

Schuetz *et al.* [83] desenvolveram um modelo para resolver o problema de coloração de grafos inspirado em conceitos de mecânica estatística. Sua técnica utiliza *Graph neural networks* [84] e o modelo de Potts [85].

O modelo de Potts [85] descreve interações em um reticulado cristalino através de *spins*. O modelo é utilizado para estudar o 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 de 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.* [86] desenvolveram uma abordagem de busca local baseada em aprendizado por reforço 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 aprendizado por reforço nas possíveis soluções do problema. O algoritmo começa com uma solução aleatória do espaço de busca. Em cada iteração, o algoritmo busca uma solução vizinha melhor de acordo com uma função de avaliação. A busca é encerrada quando uma coloração válida é obtida ou quando não existem soluções vizinhas melhores, encontrando assim uma solução ótima local.

Além destes trabalhos, é possível encontrar na literatura outras aplicações de aprendizado de máquina para a resolução do problema de coloração de grafos que utilizam técnicas e conceitos pouco explorados no contexto da alocação de registradores. Por exemplo, o trabalho de Shen *et al.* [87] utiliza conceitos de programação linear para abstrair o problema de coloração de grafo e aplicar aprendizado de máquina. Ainda, a pesquisa de Khakhulin *et al.* [88] utiliza aprendizado por reforço para resolver o problema de decomposição em árvore, o qual, ao ser resolvido, permite a solução da coloração de grafos em tempo linear.

5 RIGSET-UEL

Um dos principais desafios enfrentados por pesquisadores na aplicação de aprendizado de máquina na alocação de registradores é a falta de um conjunto de dados apropriado para treinar modelos voltados para o problema. Consequentemente, os pesquisadores da área têm recorrido a métodos alternativos para treinamento, como o desenvolvimento de um programa intermediário entre o modelo e o LLVM ou a geração de grafos aleatórios, conforme apresentado no Capítulo 4.

No entanto, os grafos aleatórios podem não simular fielmente as características de diferentes programas, limitando assim o potencial de aprendizado dos modelos de aprendizado de máquina. Além disso, outro fator importante a ser considerado no treinamento de modelos para alocação de registradores é o custo de *spill* das variáveis. A geração de custos de *spill* aleatórios pode não ser compatível com os valores encontrados em códigos reais. Ainda, a falta de conjuntos de dados relacionados ao problema pode ser um problema mesmo fora da área de aprendizado de máquina, servindo como um obstáculo no ensino da alocação de registradores e no desenvolvimento de novos alocadores.

Para resolver esse problema e auxiliar pesquisas futuras, é apresentado neste capítulo o RigSet-UEL, um conjunto de dados para suprir a falta de dados de treinamento especializados para alocação de registradores e incentivar novos estudos sobre a aplicação de aprendizado de máquina nesse contexto. Até onde foi verificado na literatura, RigSet-UEL é o primeiro conjunto de dados especializado para esse tema. O conjunto de dados proposto representa fielmente códigos reais e seus possíveis custos de *spill*, permitindo que algoritmos de aprendizado de máquina desenvolvam novas heurísticas e abordagens eficazes para a alocação de registradores.

Para formar conjunto de dados, foram coletados códigos de diversas fontes, garantindo a representatividade de diferentes tipos de códigos. A estrutura dos grafos foi elaborada para facilitar o seu uso com GNNs, redes neurais especializadas para grafos (Seção 3.3). Além disso, os dados apresentados nos grafos permitem o cálculo de custo de *spill* de variáveis, permitindo que modelos de aprendizado de máquina possam aprender sobre decisões de *spill*.

Os grafos são gerados a partir de uma representação intermediária independente de máquina, possibilitando a utilização do RigSet-UEL para diferentes arquiteturas. No entanto, a organização e as ferramentas oferecidas pelo conjunto de dados permitem sua fácil adaptação para diferentes conjuntos de registradores com variadas restrições. Apesar de estar inicialmente em formato PBQP, os dados apresentados nos grafos podem ser efetivamente aproveitados em conjunto com outras técnicas de alocação de registra-

dores, como Linear Scan, coloração de grafos, entre outras. Ainda, o conjunto de dados também pode ser utilizado em ambientes educacionais, fornecendo exemplos reais para o aprendizado prático da alocação de registradores e minimização de código de *spill*.

5.1 Gerador de Grafos de Interferência

Para construir o grafo de interferência de um código, é analisada a sua representação intermediária durante o processamento, extraindo informações sobre os intervalos de vida e o custo de *spill* de cada variável. Para criar os grafos de interferência do código, foi desenvolvido um gerador de grafos, fornecido em conjunto com o conjunto de dados. Este gerador permite a adição de grafos de interferência a partir de arquivos C/C++ no conjunto de dados. Ele também permite a regeneração dos grafos do conjunto de dados com diferentes parâmetros, como a inclusão ou personalização de vetores e matrizes de custo. Essas funcionalidades do gerador são essenciais para garantir a flexibilidade e adaptabilidade do conjunto de dados a diferentes problemas e contextos.

O processo de geração dos grafos utiliza programas em Python e ferramentas do LLVM, o qual é uma estrutura de compilador projetada para fornecer uma infraestrutura flexível para o desenvolvimento de novos compiladores e pesquisa [89]. Primeiramente, os códigos foram compilados com o *front-end* do clang [90], resultando em arquivos de representação intermediária (IR) do LLVM. A IR é uma linguagem de programação de baixo nível independente de máquina e legível por humanos, usada como uma etapa intermediária dentro da estrutura do compilador LLVM. Os grafos gerados são armazenados em arquivos JSON. A Figura 46 mostra o fluxo do processo de geração dos grafos.

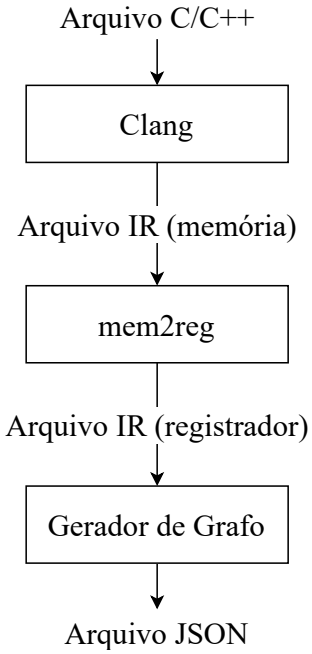


Figura 46 – Sequência de geração de grafos.

Normalmente, a IR não representa as variáveis como registradores virtuais, representando as variáveis como referências de memória, semelhante à forma que valores que sofreram *spill* são tratados. Logo, essa representação não é adequada para a criação de grafos de interferência. Por essa razão, é aplicada a passagem de transformação *mem2reg*, gerando código de representação intermediária com referências a registradores virtuais [91]. Assim, o gerador de grafos em Python pode percorrer os arquivos IR, identificando os registradores virtuais no código e coletando dados de cada um, como o tipo de valor armazenado, as linhas onde são usados, a profundidade nas estruturas de repetição de cada uso, entre outros. Com base nestas informações, o programa é capaz de gerar os tempos de vida de cada variável, identificar as interferências entre os tempos de vida e, consequentemente, construir os grafos de interferência de cada função no arquivo de código.

Para criar os grafos, o gerador identifica inicialmente o código de cada função dentro do arquivo, uma vez que cada função é representada por um grafo diferente. Em seguida, o gerador extrai os registradores virtuais das funções, os quais são sempre iniciados com o caractere '%' seguido por um número inteiro ou um ponto, como "%1" ou "%.1". Dessa forma, o programa também pode identificar a quantidade de usos de cada registrador virtual. Além disso, na primeira aparição do registrador virtual, ou seja, em sua definição, o tipo do valor armazenado é indicado no código.

Para identificar a profundidade de uso de cada registrador, foi aproveitada a estrutura SSA (*Static Single Assignment*) [92, 93] do LLVM. SSA significa que cada registrador virtual recebe apenas uma única atribuição de valor. Devido a esta característica, é necessário um tratamento especial em casos de desvios e estruturas de repetição no código, pois uma atribuição pode receber valores de diferentes registradores, dependendo do bloco de código precedente. Essas atribuições são chamadas de nós-ϕ e possuem uma notação especial no IR.

A Figura 47 mostra um nó-ϕ em uma estrutura de repetição. Ao entrar no laço, o nó-ϕ recebe o valor de i_1, que vem do bloco anterior ao laço. Nas demais interações do laço, o nó-ϕ recebe o valor de i_2, do bloco interno do laço. Como no SSA cada variável pode ser definida apenas uma vez, a variável de controle da estrutura de repetição é um nó-ϕ. A partir dos nós-ϕ, é possível verificar quais blocos estão em um laço e, consequentemente, identificar a profundidade de uso de cada registrador virtual.

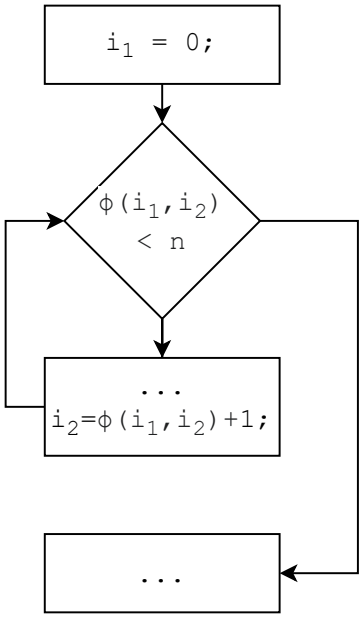


Figura 47 – Nó-ϕ em uma estrutura de repetição.

5.2 Estrutura dos Grafos Extraídos

A estrutura dos arquivos de grafo foi projetada para facilitar sua usabilidade e compreensão, além disso, são fornecidas nos vértices informações extraídas do código-fonte de cada registrador virtual correspondente aos vértices do grafo. Essas informações permitem o cálculo do custo de *spill* do registrador virtual conforme apresentado na Seção 2.4.

Cada grafo contém um objeto em que as chaves são os nomes das funções, e os valores são seus respectivos grafos. Os grafos consistem em objetos JSON separados em vértices (*nodes*) e arestas (*edges*). O valor relacionado à chave *nodes* é um objeto que contém todos os vértices, cada um associado ao nome do registrador virtual vinculado a uma variável da representação intermediária no código gerado pelo LLVM. Os vértices possuem as seguintes chaves e valores:

- ***type***: Indica o tipo de valor armazenado no registrador. Os tipos são:
  - ***i8-64***: Valores inteiros classificados pelo tamanho, que podem ser de 8, 16, 32 e 64 bits;
  - ***float***: Número de ponto flutuante com 32 bits;
  - ***double***: Número de ponto flutuante duplo com 64 bits;
  - ***ptr***: Ponteiro para um endereço ou rótulo;
  - ***unidentified***: Quando o tipo é não identificado.

- ***uses***: Um vetor com tamanho igual ao número de usos do registrador virtual. Cada valor indica um número de instrução na IR;

- ***uses deepness***: Um vetor com uma dimensão igual ao tamanho de *uses*. Os valores no vetor se referem à profundidade nas estruturas de *loop* de cada uso do registrador virtual. Por exemplo, se um valor não estiver dentro de nenhum *loop*, seu valor de profundidade será zero; se estiver dentro de um *loop*, sua profundidade será um; se estiver dentro de dois *loops* aninhados, será dois, e assim por diante;

- ***cost array***: Um vetor de custos para cada opção de alocação. Ele fornece os custos de associar o registrador virtual a outros registradores virtuais, juntamente com o custo de armazenar o valor. O programa de geração de grafos oferece uma opção para omitir o vetor de custos do arquivo de saída, gerando grafos de interferência simples. Por padrão, o vetor contém 8 registradores com custo de alocação 0.

O valor relacionado à chave *edges* é um vetor de arestas. Cada aresta contém os atributos *node 1* e *node 2*, que indicam a interferência representada pela aresta, sendo que a ordem dos vértices não importa. As arestas também incluem a matriz de custos como um terceiro atributo. Assim como o *cost array*, é possível criar grafos sem a matriz de custos nas arestas.

Esta representação do grafo e a organização das informações foram adotadas porque permitem fácil adaptação para o seu uso no contexto de aprendizado de máquina, visto que redes neurais especializadas para grafos utilizam grafos no formato de listas, onde cada elemento do grafo contém um conjunto próprio de informações relacionadas (atributos), assim como apresentado na Seção 3.3.

O Código 4 mostra um exemplo de um grafo representado no formato JSON. Cada grafo é um objeto JSON cuja chave é o nome da função no código IR, "@main" nesse exemplo. O objeto do grafo é formado por dois objetos, um para os vértices (chave "nodes") e outro para as arestas (chave "edges"). O objeto dos vértices contém um objeto para cada registrador virtual, os quais contêm os atributos do registrador virtual e utilizam o seu nome no código como chave. O objeto das arestas é formado por uma lista de arestas, onde cada item contém uma matriz de custo e dois vértices ligados pela aresta, os quais são identificados pelo nome do registrador virtual no código IR.

O principal objetivo dessa organização dos grafos é manter a versatilidade do conjunto de dados, possibilitando sua utilização com várias técnicas de alocação de registradores além do PBQP.

Os grafos PBQP podem ser empregados como grafos de interferência básicos para a alocação de registradores baseada em coloração de grafos simples. O gerador de grafos oferece uma opção para omitir o *cost array* e a *cost matrix* do grafo, criando efetivamente

```
"@main" :   {
    "nodes" : {
        "%1" : {
            "type" : "i8",
            "uses" : [1,2,3,4,6],
            "uses deepness" : [1,1,2,3,4],
            "cost array" : [0,0,0,9]
        },
        "%2" : {
            "type" : "ptr",
            "uses" : [5,6,8,11],
            "uses deepness" : [1,2,2,1],
            "cost array" : [0,0,0,6]
        }
    },

    "edges" : [
        {
            "node 1" : "%1",
            "node 2" : "%2",
            "cost matrix" : [["Infinity",0,0,9],
                             [0,"Infinity",0,9],
                             [0,0,"Infinity",9],
                             [6,6,6,15]]
        }
    ]
}
```

Código 4 – Exemplo de um grafo de função em um arquivo de saída JSON.

um grafo de interferência básico (não-PBQP). Além disso, o conjunto de dados pode ser utilizado em técnicas de alocação que não são baseadas em grafo, como o Linear Scan (Seção 2.2). O atributo *uses* pode ser utilizado para definir intervalos de vida conforme a entrada do Linear Scan.

Usuários têm a flexibilidade de criar funções de custo de *spill* baseadas em vários atributos, como *type*, *uses* e *uses deepness*. Essa capacidade permite a aplicação de diferentes heurísticas de decisões de *spill* com o conjunto de dados, facilitando testes e desenvolvimento de novos métodos para calcular os custos de *spill* de variáveis.

O gerador de grafos oferece opções adicionais para adaptar o conjunto de dados conforme as necessidades do usuário. Por padrão, um arquivo de saída pode conter múltiplos grafos, um para cada função no arquivo de entrada, mas é possível especificar que cada grafo gerado seja armazenado em um arquivo de saída separado. Conforme mostrado na Figura 4, o *cost array* e a *cost matrix* têm valores genéricos por padrão, para manter os grafos independentes de arquitetura. Portanto, o usuário pode especificar o número de registradores e seus pesos conforme necessário, o que permite simular várias arquiteturas de processadores. Além disso, o gerador de grafos permite ao usuário adicionar novos grafos ao conjunto de dados com base em seus próprios exemplos de código.

5.2.1 Exemplo de Grafo

Nesta subseção é apresentado um exemplo simples de transformação de código em um grafo no conjunto de dados. O Código 5 mostra um exemplo de código C simples, o qual gera o grafo presente na Figura 48.

```
1  void main(){
2      int m[10][10];
3      int n = 10;
4      for(int i = 0; i < n; i++){
5          for(int j = 0; j < n; j++){
6              m[i][j] = 0;
7          }
8      }
9      int r = m[0][0];
10 }
```

Código 5 – Exemplo de um código C simples.

A Figura 48 mostra o grafo de interferência gerado a partir do código anterior.

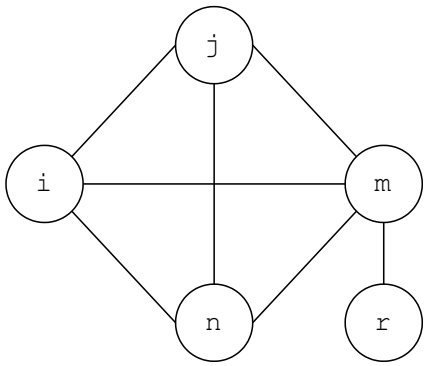


Figura 48 – O grafo de interferência do código C.

O Código 6 apresenta o grafo de interferência no formato JSON especificado na Seção 5.2 gerado a partir do Código 5. O Código 6 descreve os mesmos vértices e arestas apresentados na Figura 48. É importante notar que este não é o grafo resultante do processo de geração de grafos empregado no conjunto de dados. Para manter o exemplo simples, a etapa de representação intermediária foi omitida. O grafo de interferência foi construído com o mesmo processo, mas considerando diretamente as variáveis do código-fonte, em vez de registradores virtuais da representação intermediária. Para manter a simplicidade do exemplo, a opção de excluir vetores e matrizes de custo também foi utilizada.

```
"main" :   {
    "nodes" : {
        "n" : {
            "type" : "i8",
            "uses" : [2,3,4],
            "uses deepness" : [1,2,3]
        },
        "i" : {
            "type" : "i8",
            "uses" : [3,5],
            "uses deepness" : [2,3]
        },
        "j" : {
            "type" : "i8",
            "uses" : [4,5],
            "uses deepness" : [3,3]
        },
        "m" : {
            "type" : "ptr",
            "uses" : [1,5,6],
            "uses deepness" : [1,3,1]
        },
        "r" : {
            "type" : "i8",
            "uses" : [6],
            "uses deepness" : [1]
        }
    },
    "edges" : [
        {
            "node 1" : "n",
            "node 2" : "m"
        },
        {
            "node 1" : "n",
            "node 2" : "i"
        },
        {
            "node 1" : "n",
            "node 2" : "j"
        },
        {
            "node 1" : "i",
            "node 2" : "m"
        },
        {
            "node 1" : "i",
            "node 2" : "j"
        },
        {
            "node 1" : "j",
            "node 2" : "m"
        },
        {
            "node 1" : "m",
            "node 2" : "r"
        }
    ]
}
```

Código 6 – Exemplo de um grafo de função em um arquivo de saída JSON.

5.3 Organização e Estrutura do Conjunto de Dados

O RigSet-UEL está organizado pela origem das amostras de código. Ele é dividido em três subconjuntos: *Basic*, *CodeNet* e *Spec*. A Tabela 3 mostra o número de arquivos e o número de grafos de interferência de funções em todo o conjunto de dados e em cada subconjunto. Também são apresentadas a porcentagem de grafos de cada subconjunto em relação ao número total.

Tabela 3 – Número de arquivos e grafos no conjunto de dados e subconjuntos.

Subconjunto	Número de Arquivos	Número de Grafos	Porcentagem de Grafos
Basic	1555	3549	14,38%
CodeNet	3939	18939	76,71%
Spec	395	2200	8,91%
Total	5889	24688	

O subconjunto *Basic* contém grafos de código extraídos de repositórios de código aberto do GitHub. Ele consiste principalmente em códigos C/C++ de exercícios de lógica de programação e exemplos de cursos de programação. Desta forma, a maioria dos códigos do subconjunto consiste em demonstrações de elementos básicos de programação, como estruturas de repetição, condicionais, vetores, matrizes e recursão.

O subconjunto *CodeNet* contém grafos de amostras de código do conjunto de dados CodeNet [94], o qual é um conjunto de dados em larga escala com aproximadamente 14 milhões de amostras de código, cada uma das quais é uma solução proposta para um dos aproximadamente 4000 problemas de codificação. As amostras de código são obtidas a partir de submissões nos sites de juízes online como o AIZU Online Judge e o AtCoder. As amostras de código estão escritas em mais de 50 linguagens de programação, especialmente C++, C, Python e Java [94]. Apenas uma submissão de código por problema foi selecionada. Essa decisão foi tomada porque, apesar de serem submissões de diferentes usuários, as soluções para o mesmo problema muitas vezes compartilham estruturas de código muito semelhantes. Para evitar adicionar grafos de interferência redundantes ao conjunto de dados, uma submissão aleatória em C ou C++ foi escolhida para cada problema.

Por fim, o subconjunto *Spec* é composto por códigos extraídos dos benchmarks SPEC CPU 2017 [95]. Apenas alguns arquivos de código dos benchmarks foram selecionados devido a muitos arquivos serem incompatíveis com o processo de geração de grafos empregado. Muitos arquivos continham definições ou dependências que os tornavam compiláveis apenas durante a construção do *benchmark*. Por esse motivo, foram selecionados apenas os arquivos que podiam ser compilados individualmente. Ao contrário dos outros subconjuntos, *Spec* contém subdivisões baseadas nos benchmarks de origem do código.

A Tabela 4 apresenta as subdivisões (benchmarks) do subconjunto juntamente com seus respectivos números de arquivos e grafos.

Tabela 4 – Número de arquivos JSON e grafos em cada subconjunto de *benchmark Spec*.

Benchmark	Número de Arquivos	Número de Grafos
500.perlbench_r	6	28
502.gcc_r	29	547
505.mcf_r	20	58
507.cactuBSSN_r	8	10
511.povray_r	1	3
519.lbm_r	2	21
521.wrf_r	15	86
526.blender_r	177	581
527.cam4_r	6	6
531.deepsjeng_r	38	216
541.leela_r	8	26
544.nab_r	13	153
557.xz_r	24	28
600.perlbench_s	3	14
602.gcc_s	12	248
605.mcf_s	10	29
628.pop2_s	20	124
999.specrand_ir	2	22

Adicionalmente, o conjunto de dados inclui os arquivos IR que foram usados para gerar os grafos de interferência. Esses arquivos estão organizados da mesma maneira que os subconjuntos de grafos. O objetivo de manter os arquivos IR juntamente com os grafos no conjunto de dados é permitir que os grafos sejam recriados usando diferentes opções de geração, conforme mencionado na Seção 5.2.

5.3.1 Informações Quantitativas do RigSet-UEL

Nesta subseção, são apresentadas informações quantitativas do conjunto de dados RigSet-UEL, ilustrando as distribuições de números de nós, arestas e graus dos nós nos grafos. A Tabela 5 exibe a quantidade de grafos em todo o conjunto de dados e seus subconjuntos, categorizados pelo número de vértices que se enquadram em intervalos específicos.

Tabela 5 – Número de grafos por quantidade de vértices no conjunto de dados e subconjuntos.

Número de vértices	Basic	CodeNet	Spec	Total
0 - 15	1749	9385	897	12031
16 - 31	804	3015	397	4216
32 - 47	403	1464	215	2082
48 - 63	188	1233	118	1539
64 - 79	120	717	106	943
80 - 95	81	557	78	716
96 - 111	50	457	61	568
112 - 127	50	328	35	413
$\geq$ 128	104	1783	293	2180

A Figura 49 apresenta um histograma dos dados totais do conjunto de dados apresentados na Tabela 5.

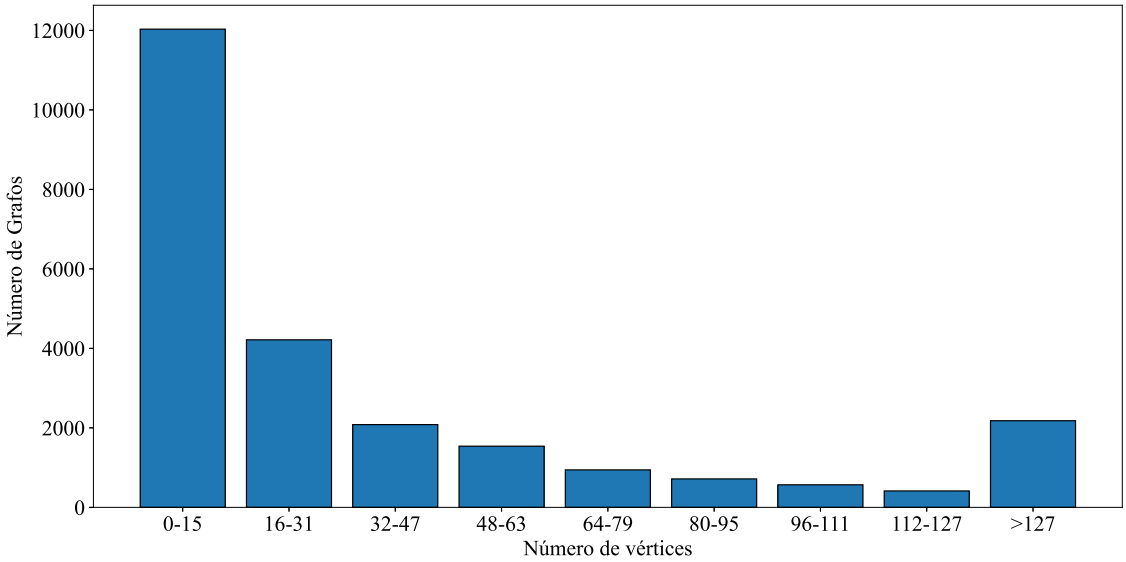


Figura 49 – Número de grafos por quantidade de vértices no conjunto de dados.

A Tabela 5 e a Figura 49 demonstram que a maioria dos grafos foi gerada a partir de funções pequenas, visto que existe um número maior de grafos com poucos vértices no conjunto de dados. Ainda, observa-se que essa distribuição se manteve semelhante entre diferentes subconjuntos, apesar de suas diferentes origens, indicando que a prevalência de funções pequenas é comum mesmo em diferentes contextos. Essa consistência é importante, pois aponta para uma boa representação da distribuição real do tamanho dos códigos que são fornecidos ao compilador.

A Tabela 6 exibe a quantidade de grafos em todo o conjunto de dados e seus subconjuntos, categorizados pelo número de arestas que se enquadram em intervalos específicos.

Tabela 6 – Número de grafos por quantidade de arestas no conjunto de dados e subconjuntos.

Número de arestas	Basic	CodeNet	Spec	Total
0 - 15	1139	5941	489	7569
16 - 47	571	3064	364	3999
48 - 79	330	1469	196	1995
80 - 127	288	1118	158	1564
128 - 191	246	950	111	1307
192 - 255	193	623	99	915
256 - 511	345	1600	178	2123
512 - 1023	214	1530	215	1959
$\geq$ 1024	223	2644	390	3257

A Figura 50 apresenta um histograma dos dados totais do conjunto de dados apresentados na Tabela 6.

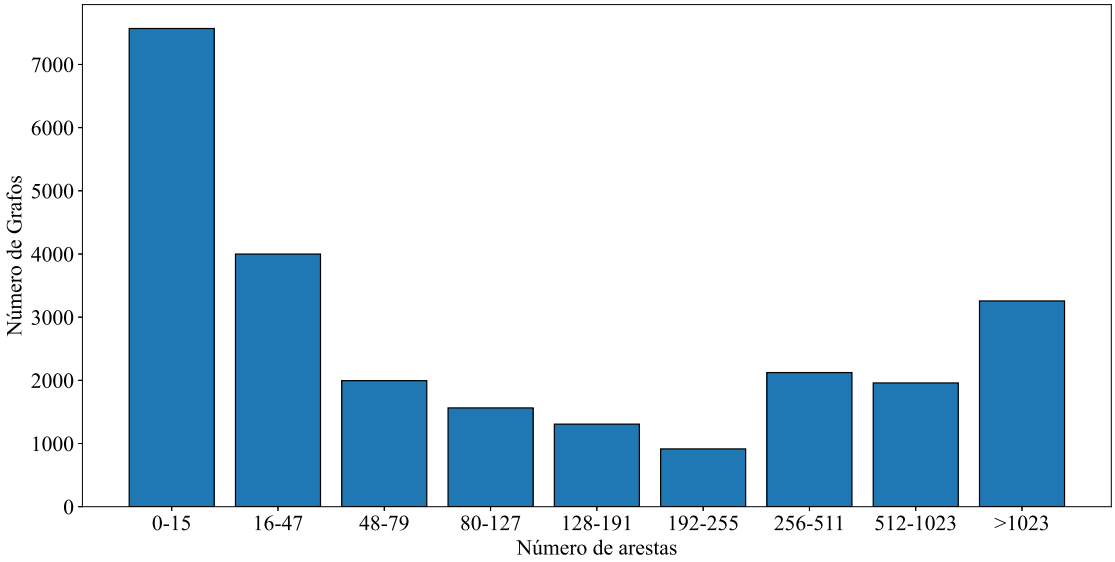


Figura 50 – Número de grafos por quantidade de arestas no conjunto de dados.

A Tabela 6 e a Figura 50 reforçam as observações feitas sobre a Tabela 5 e a Figura 49. O conjunto de dados contém um número menor de grafos com muitas arestas, distribuições que se mantêm consistentes entre os subconjuntos. Essa semelhança é outra indicação de que o conjunto de dados apresenta uma boa representação de códigos reais. Além disso, é possível concluir que, em geral, os grafos no conjunto de dados são mais densos do que esparsos.

A Tabela 7 exibe a quantidade de vértices em todos os grafos do conjunto de dados e seus subconjuntos, categorizados pelos graus dos vértices que se enquadram em intervalos específicos.

Tabela 7 – Número de vértices por grau dos vértices no conjunto de dados e subconjuntos.

Número de vértices	Basic	CodeNet	Spec	Total
0 - 2	2127	10915	768	13810
3 - 5	813	5122	779	6714
6 - 8	381	1814	429	2624
9 - 11	138	672	96	906
12 - 14	55	238	48	341
15 - 17	10	88	33	131
18 - 20	9	59	8	76
21 - 23	4	19	3	26
≥ 24	12	12	36	60

A Figura 51 apresenta um histograma dos dados totais do conjunto de dados apresentados na Tabela 7.

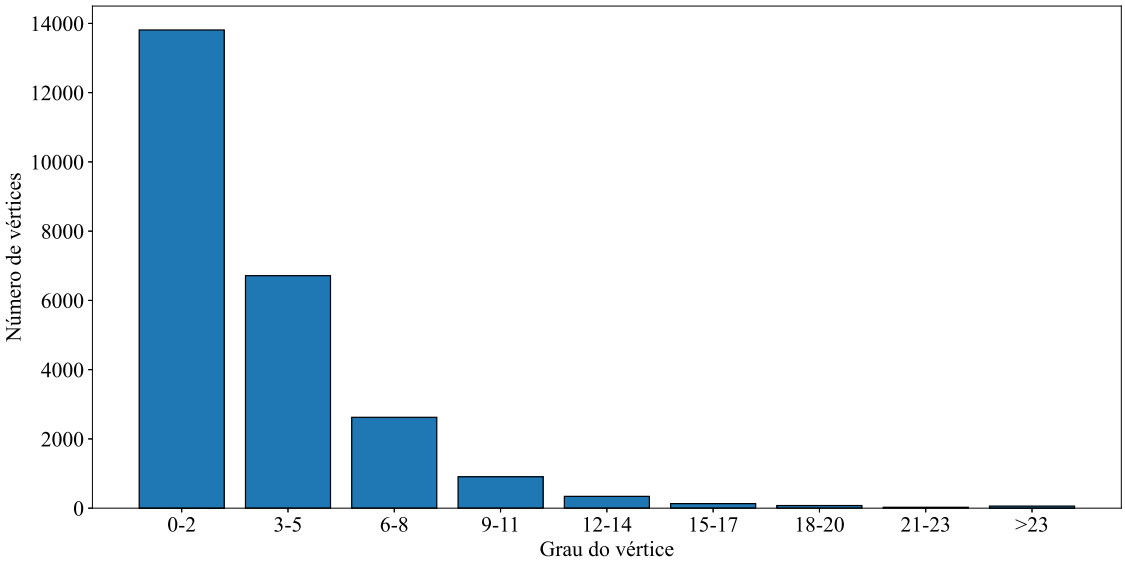


Figura 51 – Número de vértices por grau dos vértices no conjunto de dados.

Observa-se na Tabela 51 e na Figura 51 que a maioria dos vértices possuem graus baixos. Vértices com grau menor que o número de registradores disponíveis são sempre alocados. Portanto, vértices com graus elevados são necessários para criar cenários complexos de alocação, especialmente considerando que quanto maior a interferência de um nó, maior a chance de ocorrer um *spill*. Neste contexto, o número de vértices de alto grau pode parecer baixo, porém é importante considerar que grande parte dos vértices representa variáveis temporárias e auxiliares, as quais são utilizadas durante poucas operações. Além disso, mesmo um pequeno número de vértices de graus elevados em um grafo pode dificultar a alocação e gerar a necessidade de *spill*.

6 MODELOS DE APRENDIZADO DE MÁQUINA PARA ALOCAÇÃO DE REGISTRADORES

Para demonstrar o uso do conjunto de dados RigSet-UEL foram desenvolvidos modelos de aprendizado de máquina com GNN (Seção 3.3) e aprendizado por reforço (Subseção 3.1.1) para a criação de novas heurísticas de coloração para serem utilizadas na alocação de registradores, para algoritmos de coloração de grafos e PBQP. Diferentemente das abordagens apresentadas no Capítulo 4, as quais exigem modificações substanciais na estrutura do alocador, os modelos propostos buscam integrar-se diretamente aos algoritmos existentes, substituindo apenas o componente da heurística. Além disso, por substituir apenas a heurística e manter a estrutura dos algoritmos, essa abordagem elimina a necessidade de garantir a corretude do resultado da alocação, uma grande adversidade enfrentada ao elaborar modelos de aprendizado de máquina na alocação de registradores [7].

Tanto na coloração de grafos quanto no PBQP são utilizadas heurísticas para definir a ordem de remoção dos vértices do grafo de interferência. Nos modelos propostos essas heurísticas serão substituídas por uma GNN. Sempre que um vértice necessita ser removido durante o processo de alocação, os modelos desenvolvidos recebem os vértices juntamente com seus conjuntos de características e retornam um único valor para cada um. O vértice com o maior valor de saída é selecionado para remoção naquela iteração.

Considerando o cenário de aprendizado por reforço, em cada iteração o agente (GNN) escolhe um vértice (ação) que será então removido do grafo de interferência (ambiente). A única diferença em relação ao cenário tradicional é que o reforço apenas é fornecido ao agente após a solução do problema (alocação completa do grafo), e não após cada ação de remoção. Isso ocorre pois as funções de recompensa são calculadas a partir da comparação da alocação obtida em relação aos resultados ótimos de cada grafo. Os custos ótimos foram obtidos por força bruta, utilizando o algoritmo de alocação PBQP com *branch-and-bound* (Subseção 2.3.2).

Esse método permite uma avaliação controlada e isolada da heurística baseada em aprendizado de máquina em comparação com as técnicas tradicionais, uma vez que os demais componentes do alocador permanecem inalterados. Ao minimizar o custo de integração e preservar a compatibilidade com algoritmos de alocação amplamente utilizados, esta abordagem facilita a adoção prática ao mesmo tempo em que explora o potencial do aprendizado de máquina para aprimorar a qualidade da alocação.

Foram conduzidos experimentos com seis modelos, sendo três voltados para coloração de grafos e três para PBQP. Cada modelo foi treinado utilizando três funções de

recompensa distintas. Com o intuito de avaliar exclusivamente o desempenho das heurísticas, considerou-se uma arquitetura regular genérica com oito registradores. Os modelos foram comparados com o alocador PBQP proposto por Hames e Scholz (Subseção 2.3.1), com coloração otimista. Como todos os registradores possuem o mesmo peso, o critério de coloribilidade opera de maneira equivalente à *optimistic coloring* (Subseção 2.1.2) para coloração de grafos.

6.1 Modelos de Coloração de Grafo

Na coloração de grafos, os vértices são removidos e adicionados a uma pilha, sendo posteriormente reintroduzidos no grafo e alocados. A função da rede neural é determinar a ordem na qual os vértices são removidos do grafo e inseridos na pilha. Em cada iteração, a rede recebe o estado atual do grafo e seleciona qual dos vértices restantes será removido (e adicionado na pilha) na iteração. Após a remoção de todos os vértices, a fase de reconstrução é realizada conforme o procedimento usual, onde os vértices são retirados da pilha, adicionados novamente ao grafo e associados a um registrador. Dessa forma, após a reconstrução completa, o grafo alocado é obtido. Este processo está ilustrado na Figura 52.

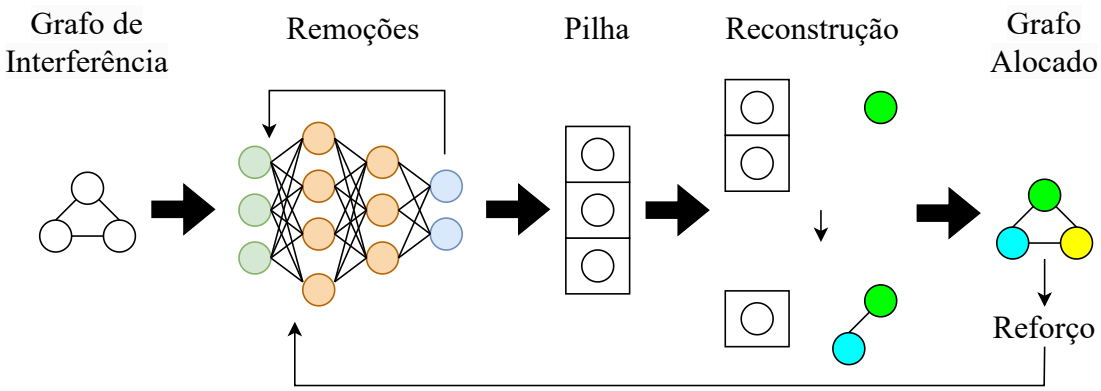


Figura 52 – Fluxograma dos modelos de coloração de grafo.

Durante o treinamento, também é necessário fornecer recompensas para permitir o aprendizado dos modelos. Após o fim da alocação, a função de recompensa é calculada com base no *spill* resultante da alocação e no resultado ótimo do grafo. Observa-se que, dessa forma, o modelo recebe uma recompensa apenas após a alocação completa do grafo, e não após cada ação, como geralmente acontece no cenário de aprendizado por reforço.

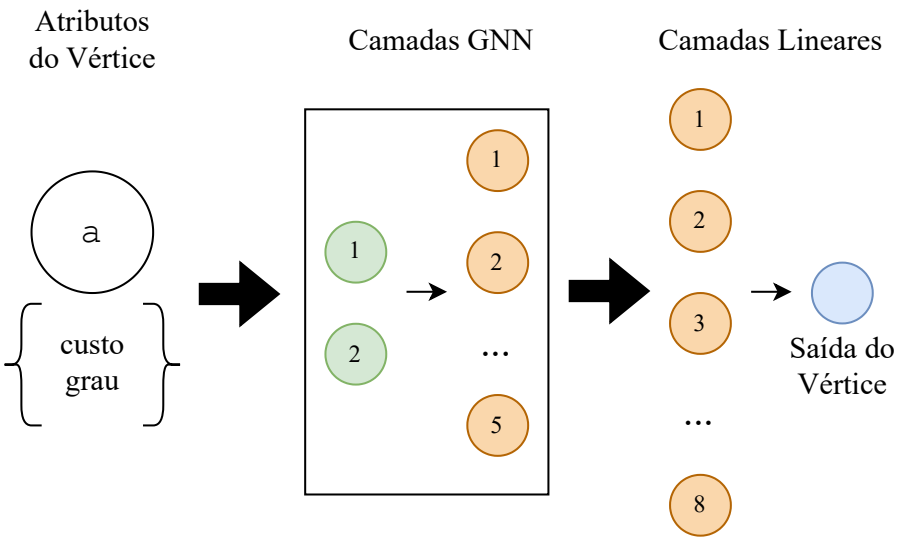


Figura 53 – Arquitetura da GNN para modelos de coloração de grafo.

Para decidir qual vértice será removido em cada interação, a GNN realiza transformações nos atributos de cada vértice. Os atributos presentes no vértice são seu grau e o custo estimado de *spill*. Com base nessas informações, a GNN retorna um único atributo para cada vértice indicando sua prioridade de remoção. Desta forma, é escolhido o vértice com o atributo de maior valor para ser removido. Essa transformação realizada pela GNN, assim como sua arquitetura, são apresentadas na Figura 53.

6.2 Modelos de PBQP

Nos modelos PBQP os vértices são alocados ao serem removidos do grafo. Preferencialmente, são realizadas remoções ótimas (R1 e R2). A partir do momento que não restam remoções ótimas possíveis, são realizadas remoções não ótimas (RN). A ordem na qual as remoções não ótimas são realizadas pode afetar a qualidade da alocação. Nesse contexto, a GNN é responsável por realizar reduções N. Após a execução de todas as remoções possíveis pelos métodos R1 e R2, a rede determina qual vértice será removido em cada iteração. Quando removido, o vértice é alocado em um registrador livre se existir algum disponível. Caso contrário, o vértice sofre *spill*. A Figura 54 ilustra esse processo. Assim como nos modelos de coloração de grafos, a recompensa é calculada e fornecida apenas após a alocação completa do grafo.

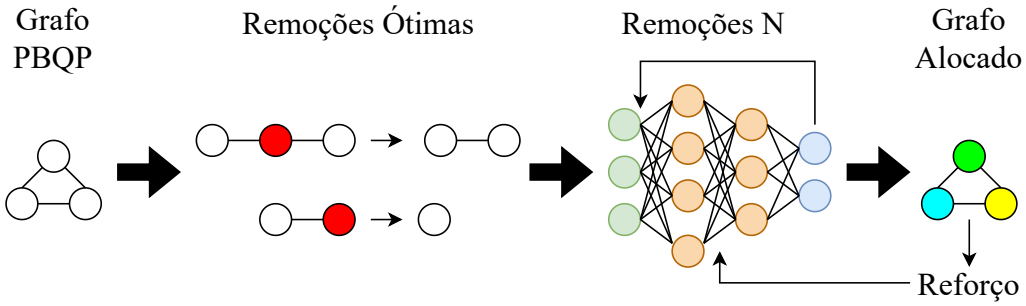


Figura 54 – Fluxograma dos modelos de PBQP.

Assim como nos modelos de coloração de grafos, a GNN realiza transformações nos vértices, recebendo seus atributos e retornando um único para cada. O vértice com o maior atributo após essa transformação é então removido e alocado. Entretanto, além do grau do vértice e do custo de *spill*, a rede neural considera também os pesos de alocação para cada um dos oito registradores, totalizando dez características. Inicialmente, todos os pesos são definidos como zero, podendo assumir o valor infinito quando um vértice adjacente é removido (e, consequentemente, alocado). Para a rede neural, o valor infinito é representado numericamente por dois. A transformação e a arquitetura da GNN são apresentadas na Figura 55.

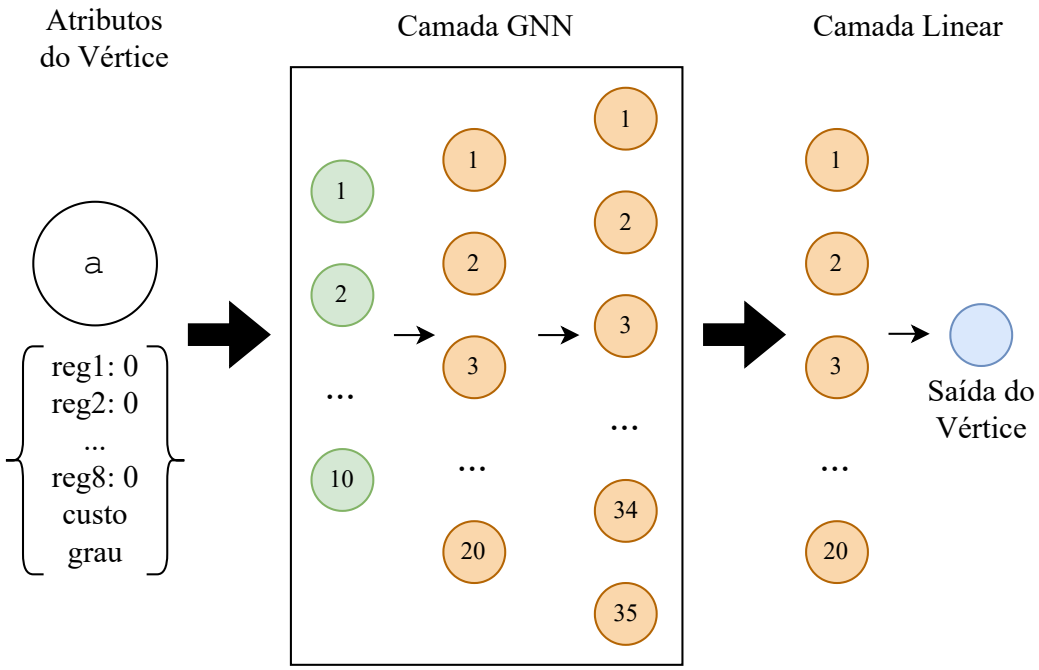


Figura 55 – Arquitetura da GNN para modelos de PBQP.

6.3 Treinamento

Os modelos foram implementados utilizando a biblioteca PyTorch [96] e treinados por 300 épocas. Como foram considerados oito registradores disponíveis, grafos com até oito vértices foram filtrados por apresentarem alocações triviais no cenário proposto. Em seguida, 20% dos grafos foram selecionados aleatoriamente para compor o conjunto de teste, utilizado na avaliação do desempenho dos modelos. O conjunto de treinamento é composto por 12555 grafos, enquanto o conjunto de teste contém 3139 grafos.

Com relação à taxa de aprendizado (lr), a variável que indica o quanto os pesos da rede neural são ajustados em uma determinada época [97], foi definida uma função para determinar seu valor com base na época de treinamento. Considerando lr_0 a taxa de aprendizado inicial, lr_{decay} a taxa de decaimento da taxa de lr e $lr_{\min}$ o valor mínimo que lr pode atingir, a taxa de aprendizado lr para a época n é dada pela seguinte equação:

$$lr_n = \max\left(lr_0 \cdot \frac{1}{1 + lr_{\text{decay}} \cdot n},\ lr_{\min}\right) \tag{6.1}$$

Foram utilizados os seguintes valores: $lr_0 = 0{,}1$, $lr_{decay} = 0{,}5$ e $lr_{\min} = 0{,}1$. Dessa forma, a taxa de aprendizado inicia com valores elevados, consolidando rapidamente o aprendizado nas épocas iniciais. Posteriormente, seu valor é reduzido, de modo que as épocas finais realizem apenas ajustes sutis, evitando que o aprendizado previamente adquirido seja apagado.

Para controlar o equilíbrio entre *exploration versus exploitation* (Seção 3.1.1), um valor aleatório entre 0 e 1 é gerado a cada decisão e comparado com a variável ε. Se o valor gerado for $\geq \varepsilon$, a decisão é tomada pela rede neural; caso contrário, uma decisão aleatória é adotada. Considerando $\varepsilon_{\text{decay}}$ a taxa de decaimento de ε, e $\varepsilon_{\min}$ o valor mínimo que ε pode atingir, o valor de ε na época n é definido pela seguinte equação:

$$\varepsilon_n = \max\left(\varepsilon_{n-1} \cdot \varepsilon_{\text{decay}},\ \varepsilon_{\min}\right) \tag{6.2}$$

Foram considerados os seguintes parâmetros: $\varepsilon_0 = 1$, $\varepsilon_{decay} = 0{,}99$ e $\varepsilon_{\min} = 0{,}1$. Assim, o treinamento é iniciado com uma alta taxa de exploração, a qual é reduzida gradualmente ao longo das épocas.

6.4 Spill Extra

Para a avaliação de resultados não ótimos, define-se o cálculo do *spill extra*, que representa o custo adicional de *spill* no resultado da alocação em comparação com o

resultado ótimo. O custo de *spill* é determinado com base na heurística de Chaitin (Seção 2.4). O cálculo do *spill extra* é dado pela seguinte equação:

$$spill\ extra = \frac{spill\ resultante - custo\ otimo}{spill\ total - custo\ otimo} \tag{6.3}$$

O *spill total* corresponde à soma do custo de *spill* de todos os vértices do grafo. Esse valor é utilizado para normalizar o resultado entre diferentes grafos.

6.5 Recompensa

Foram definidas e nomeadas as seguintes funções de recompensa para realizar o treinamento dos modelos:

- ***SpillExtra***: Retorna um valor positivo caso o resultado seja ótimo, ou um valor negativo caso não seja. Quando uma alocação é ótima, seu *spill extra* é igual a 0, de modo que a recompensa é definida por:

$$R = \begin{cases} 1, & \text{se } spill\ extra = 0 \\ -spill\ extra, & \text{caso contrário} \end{cases} \tag{6.4}$$

  Esta é uma função simples, projetada para incentivar a busca por soluções ótimas, ao mesmo tempo que minimiza a penalidade para resultados subótimos com base na quantidade de *spill* resultante;

- ***UltimaDif***: Retorna a diferença do *spill extra* do grafo entre a época anterior e a atual. A função é expressa por:

$$R = spill\ extra_{t-1} - spill\ extra_t \tag{6.5}$$

  Caso o *spill extra* atual seja inferior ao da época anterior, a recompensa será positiva. Esta função foi concebida para incentivar a redução contínua do *spill*;

- ***MelhorDif***: Semelhante à *UltimaDif*, porém, em vez de comparar com a época anterior, a comparação é realizada com o menor valor de *spill extra* obtido até o momento durante o treinamento. Considerando $melhor = \min_{1 \leq i < t} spill\ extra_i$ o melhor (menor) *spill extra* alcançado até a época t. A recompensa é dada por:

$$R = melhor - spill\ extra_t \tag{6.6}$$

  Esta função busca estimular a melhoria contínua em relação ao melhor resultado previamente obtido.

Foram testadas também variações dessas funções, como retornar 1 e -1 para *UltimaDiff* e *UltimaDiff* de acordo com a redução ou aumento da quantidade de *spill extra* e definição de limiares aceitáveis de *spill extra*, que forneceriam 0 como recompensa. Porém, como essas variações não obtiveram diferenças significativas em seus resultados, foram selecionadas as três funções apresentadas para a realização do experimento. Essas funções foram escolhidas pois abrangem todos os focos de treinamento testados, não dependem de limiares arbitrários de *spill extra* aceitáveis e, em teoria, permitem um aprendizado mais sensível ao retornarem valores relacionados ao *spill extra* em vez de simplesmente 1 e -1.

6.6 Resultados

A Tabela 8 apresenta os resultados dos seis experimentos realizados, juntamente com o resultado de referência obtido pelo alocador de Hames e Scholz (Subseção 2.3.1). Para cada resultado, são apresentadas as seguintes métricas: a porcentagem de grafos com soluções ótimas; a média do *spill extra* para os casos não ótimos, expressa em percentual; e a média do número de nós e arestas nos grafos com resultados não ótimos.

Os experimentos foram conduzidos em um computador equipado com processador AMD Ryzen 5 PRO 5650G com gráficos integrados Radeon e contava com 32 GB de memória RAM. O sistema operacional utilizado foi o Ubuntu 22.04.5 LTS (64 bits). O cálculo dos resultados ótimos com *branch-and-bound* durou 22 horas, o treinamento de cada modelo durou 3 semanas e a etapa de teste durou 30 minutos para cada alocador.

Modelos		**Ótimo**	**Spill extra**	**Vértices**	**Arestas**
Hames e Scholz		35,7%	15,2%	91,6	1590,55
GC	*SpillExtra*	31,7%	23,09%	87,59	1503,05
	UltimaDif	31,7%	23,09%	87,59	1503,05
	MelhorDif	31,57%	23,93%	87,49	1500,49
PBQP	*SpillExtra*	35,3%	35,89%	92,85	1776,8
	UltimaDif	35,07%	24,72%	90,95	1576,08
	MelhorDif	35,36%	25,07%	91,28	1503,06

Tabela 8 – Resultado dos modelos propostos nos experimentos.

Os modelos baseados em PBQP alcançaram uma quantidade de resultados ótimos semelhante à obtida pelo alocador de Hames e Scholz, porém apresentaram maiores discrepâncias em termos de *spill extra*. Além disso, esses modelos apresentaram melhor desempenho quando treinados com funções de recompensa focadas na minimização do *spill*, com uma melhora de 10% na média do *spill extra*. Os modelos baseados em coloração de grafos demonstraram resultados consistentes independentemente do tipo de recompensa utilizada. Apesar de obterem o menor número de resultados ótimos, apresentaram uma média de *spill extra* inferior a dos modelos PBQP.

Os tamanhos dos grafos com resultados subótimos foram coletados para verificar discrepâncias nos casos de teste em que cada modelo apresentou dificuldade na alocação. A média do número de vértices e arestas nos resultados subótimos foi semelhante para todos os alocadores, indicando consistência nos grafos com desempenho inferior ao ótimo ao longo dos diferentes experimentos.

Observa-se que, embora próximos, os modelos apresentaram desempenho inferior ao do alocador baseado em *optimistic coloring*, especialmente em relação à quantidade de *spill extra*. Esse resultado sugere que heurísticas baseadas em aprendizado de máquina possuem potencial para aplicação na alocação de registradores, embora ainda não sejam uma alternativa à altura de heurísticas consolidadas. Portanto, espera-se que trabalhos futuros possam basear-se nesses modelos para desenvolver novas técnicas, facilitando o processo de desenvolvimento de heurísticas para a alocação de registradores e contribuindo para a geração de código mais otimizado.

O uso de aprendizado de máquina para criação de heurísticas para a alocação de registradores pode ser particularmente vantajoso em cenários ou arquiteturas específicas, nos quais modelos baseados em aprendizado de máquina podem explorar características particulares que heurísticas gerais não capturam adequadamente, e para os quais heurísticas especializadas ainda não foram amplamente desenvolvidas, devido à complexidade do processo de sua criação. Outra alternativa interessante seria utilizar o aprendizado de máquina para tomar decisões de *spill*, em vez de realizar a coloração. Dessa forma, o modelo teria um papel mais especializado, o que poderia simplificar o processo de aprendizado.

7 CONCLUSÃO

A utilização de modelos de aprendizado de máquina para otimizar a alocação de registradores é um tema promissor que surgiu recentemente na literatura, com o potencial de melhorar o desempenho de códigos gerados por compiladores. Entretanto, uma adversidade enfrentada por pesquisas na área é a escassez de dados especializados para o treinamento de modelos. Por esse motivo trabalhos sobre esse tema precisaram recorrer a meios alternativos, como a geração de grafos aleatórios. Além da inconveniência durante o treinamento dos modelos, a geração de grafos aleatórios não representa códigos reais, afetando o aprendizado dos modelos. Essa adversidade foi solucionada nesse trabalho através da criação do RigSet-UEL, o primeiro conjunto de dados elaborado para o treinamento de modelos de aprendizado de máquina voltados para a alocação de registradores.

O RigSet-UEL é formado por grafos de interferência gerados a partir de códigos reais. A estrutura dos grafos é intuitiva e permite o armazenamento de diversos dados sobre as variáveis a serem alocadas. Desta forma, esses dados podem ser facilmente processados e transformados para atender aos requisitos de diferentes técnicas de alocação de registradores, mesmo considerando abstrações que não utilizam grafos, como o Linear Scan. O RigSet-UEL contém um gerador de grafos, que possibilita a expansão do conjunto de dados e sua adaptação para contextos específicos. Ainda, o RigSet-UEL pode ser útil em contextos educacionais, fornecendo exemplos práticos para o ensino da alocação de registradores, permitindo que estudantes observem os resultados de diferentes algoritmos de alocação e técnicas de minimização de *spill* em diversos grafos de interferência.

Além disso, foram desenvolvidos modelos de aprendizado de máquina baseados em GNN e aprendizado por reforço para a criação de novas heurísticas de coloração para a alocação de registradores para demonstrar o uso do RigSet-UEL. Esses modelos apresentam uma abordagem inédita de aplicar aprendizado de máquina no problema. Ao manter a estrutura de algoritmos já utilizados, os modelos não afetam a corretude da alocação, uma adversidade enfrentada por outras abordagens presentes na literatura. Apesar de não atingirem o desempenho de heurísticas consolidadas, os modelos descritos apresentam novos métodos promissores para a aplicação de aprendizado de máquina na alocação de registradores e indicam novos caminhos para futuras pesquisas na área.

O RigSet-UEL e a descrição de novas abordagens de aplicação de aprendizado de máquina na alocação de registradores facilitam e incentivam o desenvolvimento de novas pesquisas sobre o tema. Portanto, esse trabalho contribui para a criação de novas técnicas para o problema da alocação de registradores por meio de aprendizado de máquina e, consequentemente, para melhorar o desempenho de códigos gerados por compiladores.

Devido à área de pesquisa ser relativamente nova e o RigSet-UEL ser o primeiro conjunto de dados voltado para o tema, não existem métricas para avaliar a qualidade e representatividade dos grafos de interferência e, consequentemente, dos tipos de códigos que devem formar o conjunto de dados, dificultando a seleção dos códigos que seriam utilizados para criar os grafos. Isso pode ser explorado em futuros trabalhos, auxiliando a expansão do RigSet-UEL ou criação de outros conjuntos de dados.

Outra possibilidade para trabalhos futuros é a utilização de modelos generativos de inteligência artificial para obter exemplos de código que possibilitem a geração de novos grafos. Essa abordagem garante que os grafos reflitam códigos reais, em vez de simplesmente gerar grafos aleatórios diretamente. Ainda, seria interessante a adição de ferramentas que permitam ajustar os grafos para emular mais precisamente arquiteturas específicas, visto que, no momento, essa funcionalidade está limitada a definir diferentes custos de acesso para cada registrador.

TRABALHOS PUBLICADOS PELO AUTOR

O seguinte trabalho foi publicado pelo autor durante o programa de mestrado:

1. Pedro Zaffalon da Silva, Helen Cristina de Mattos Senefonte, Wesley Attrot, **Interference Graph Dataset for Machine Learning-Based Register Allocation**, IEEE ACCESS, vol. 12, pp. 157574-157586, Novembro/2024, doi: 10.1109/ACCESS.2024.3481358, (Qualis CC 2020, A3).

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] SUTTON, R. S.; BARTO, A. G. et al. *Reinforcement learning: An introduction*. [S.l.]: MIT press Cambridge, 1998. v. 1.

[4] PARMEZAN, A.; SOUZA, V. Alves de; BATISTA, G. Evaluation of statistical and machine learning models for time series prediction: Identifying the state-of-the-art and the best conditions for the use of each model. *Information Sciences*, 01 2019.

[5] SANCHEZ-LENGELING, B. et al. A gentle introduction to graph neural networks. *Distill*, v. 6, n. 9, p. e33, 2021.

[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] 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>.

[13] 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.

[14] 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.

[15] 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>.

[16] 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>.

[17] 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>.

[18] 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>.

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

[20] 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>.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] TROFIN, M. et al. *MLGO: a Machine Learning Guided Compiler Optimizations Framework*. 2021. Disponível em: <https://arxiv.org/abs/2101.04808>.

[27] SHAHZAD, H. et al. Reinforcement learning strategies for compiler optimization in high level synthesis. In: IEEE. *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. [S.l.], 2022. p. 13–22.

[28] 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.

[29] 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.

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

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

[32] X64 Architecture. Https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture?redirectedfrom=MSDN. Accessed: 2023-26-10.

[33] 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>.

[34] SCHOLZ, B.; ECKSTEIN, E. Register allocation for irregular architectures. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2002. (LCTES/SCOPES '02), p. 139–148. ISBN 1581135270. Disponível em: <https://doi.org/10.1145/513829.513854>.

[35] BURKARD, R. E. et al. *The quadratic assignment problem*. [S.l.]: Springer, 1998.

[36] 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>.

[37] 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>.

[38] 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.

[39] 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.

[40] 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>.

[41] 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>.

[42] 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>.

[43] 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.

[44] WHAT is deep learning? Https://www.ibm.com/topics/deep-learning. Accessed: 2025-02-01.

[45] WHAT is machine learning? Https://www.ibm.com/think/topics/machine-learning. Accessed: 2025-02-01.

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

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

[48] SOOFI, A. A.; AWAN, A. Classification techniques in machine learning: applications and issues. *Journal of Basic & Applied Sciences*, v. 13, p. 459–465, 2017.

[49] MAULUD, D.; ABDULAZEEZ, A. M. A review on linear regression comprehensive in machine learning. *Journal of applied science and technology trends*, v. 1, n. 2, p. 140–147, 2020.

[50] HUANG, Z.-H. et al. Survey on learning-to-rank based recommendation algorithms. Deakin University, 2016.

[51] 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>.

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

[53] SENEFONTE, H. C. de M. *Aceleração do aprendizado por reforço em sistemas com múltiplos objetivos*. Dissertação (Mestrado Acadêmico) — Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, Brasil, 2009.

[54] 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.

[55] 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>.

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

[57] 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.

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

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

[60] JAKHAR, D.; KAUR, I. Artificial intelligence, machine learning and deep learning: definitions and differences. *Clinical and experimental dermatology*, Blackwell Publishing Ltd Oxford, UK, v. 45, n. 1, p. 131–132, 2020.

[61] 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>.

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

[63] O'SHEA, K.; NASH, R. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

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

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

[66] LIN, T. et al. A survey of transformers. *AI open*, Elsevier, v. 3, p. 111–132, 2022.

[67] SCARSELLI, F. et al. The graph neural network model. *IEEE transactions on neural networks*, IEEE, v. 20, n. 1, p. 61–80, 2008.

[68] GILMER, J. et al. *Neural Message Passing for Quantum Chemistry*. 2017. Disponível em: <https://arxiv.org/abs/1704.01212>.

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

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

[71] 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>.

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

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

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

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

[76] 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.

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

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

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

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

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

[82] 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>.

[83] 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>.

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

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

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

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

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

[89] LATTNER, C.; ADVE, V. The llvm compiler framework and infrastructure tutorial. In: SPRINGER. *Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers 17*. [S.l.], 2005. p. 15–16.

[90] CLANG: A C language family frontend for LLVM. Https://clang.llvm.org/. Accessed: 2023-25-08.

[91] THE LLVM Compiler Infrastructure. Https://llvm.org. Accessed: 2023-25-08.

[92] 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>.

[93] BUCHWALD, S.; ZWINKAU, A.; BERSCH, T. Ssa-based register allocation with pbqp. In: SPRINGER. *Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*. [S.l.], 2011. p. 42–61.

[94] PURI, R. et al. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. 2021.

[95] SPEC CPU 2017 Benchmark Suite. 2017. Https://www.spec.org/cpu2017. Acesso em 1 de setembro de 2023.

[96] PASZKE, A. et al. Pytorch: an imperative style, high-performance deep learning library. In: _____. *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[97] WHAT is learning rate in machine learning? Https://www.ibm.com/think/topics/learning-rate. Accessed: 2025-02-10.

Apêndices

APÊNDICE A

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 17 September 2024, accepted 5 October 2024, date of publication 16 October 2024, date of current version 4 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3481358

APPLIED RESEARCH

Interference Graph Dataset for Machine Learning-Based Register Allocation

PEDRO ZAFFALON DA SILVA, HELEN CRISTINA DE MATTOS SENEFONTE, AND WESLEY ATTROT

Department of Computing, State University of Londrina, Londrina 86057-970, Brazil

Corresponding author: Pedro Zaffalon da Silva (pedro.zaffalon@uel.br)

This work was supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior–Brasil (CAPES)–Finance Code 001, and in part by the Superintendência de Ciência, Tecnologia e Ensino Superior (SETI) e Universidade Estadual de Londrina (PROPPG).

ABSTRACT Register allocation is an important phase in compiler optimization. Often, its resolution involves graph coloring, which is an NP-complete problem. Because of their significance, numerous heuristics have been proposed for their resolution. Heuristic development is a complex process that requires specialized domain expertise. Recently, several machine learning based approaches have been proposed to solve compiler optimization problems. Nevertheless, owing to the complexity of the problem and the lack of specialized datasets for training models applied to register allocation, few works on the topic have been produced. The deficiency in sufficient adequate test cases is a recurring issue when working with register allocation, even beyond the scope of machine learning applications. In an effort to address this problem and facilitate forthcoming research in this domain, this work presents the development of a simple code interference graph generator, and as far as we know, the first dataset for training machine learning models focused on register allocation. The dataset consists of PBQP interference graphs from real C/C++ codes, and it is easily adaptable to various register allocation techniques. Our dataset primary goal is to represent codes and potential spill costs accurately as graphs to enable machine learning algorithms to develop effective new register allocation approaches, however it can also assist in the development and improvement of more traditional register allocation techniques.

INDEX TERMS Register allocation, dataset, interference graph, machine learning, PBQP.

I. INTRODUCTION

Register allocation is a crucial compiler step responsible for mapping variables to the physical storage of a computer and determining whether each variable should reside in a register or the main memory. It is important to underscore the substantial discrepancy in access speed between registers and main memory. Therefore, reducing variable storage in memory is crucial for optimizing the performance of compiler-generated code [36]. An alternative to this problem is to use more advanced hardware [21], [22], [35], [44]. However, in many cases, replacing physical equipment is financially costly. Software-based solutions only require recompilation, without the need to replace any hardware, which is significantly less expensive. Additionally, software improvements can also extend the hardware usage time. Therefore, it is important to consider a software-based solution.

Owing to its significance in terms of code generation quality, register allocation has been extensively explored for decades. It is commonly abstracted as graph coloring, an NP-complete optimization problem [7], [13], [23], where each vertex represents the live range of a variable, and edges indicate variables that need to be stored simultaneously at some point. In this context, colors correspond to registers that need to be allocated, requiring the assignment of colors to vertices without any adjacent vertices sharing the same color.

Given the NP-complete nature of the problem, optimal solutions cannot be guaranteed by existing algorithms. Therefore, it is crucial to develop heuristics to improve

The associate editor coordinating the review of this manuscript and approving it for publication was Yiming Tang.

register allocation. Numerous heuristics have been proposed over the past 40 years [10], [13], [15], and new heuristics have been explored [14]. Nonetheless, developing heuristics is a complex process that demands expertise in highly specialized fields, encompassing compiler construction and hardware architectures [34]. Furthermore, the employed heuristics often exhibit optimization opportunities and may need to be tailored to specific architectures [42].

On the other hand, with the proliferation of machine learning [6], [19], [43] across various domains, neural network-based approaches have emerged as a promising option to enhance compiler optimization. Machine learning is an efficient method for developing heuristics for complex problems [25]. Thus, applying machine learning to register allocation could result in better heuristics than those currently available, and consequently, in the generation of more optimized code. The development of heuristics is a complex and labor-intensive process, but it can be considerably simplified with the use of machine learning, making it easier to develop more specialized allocators for specific contexts [25], [28], [34], [41].

However, applying machine learning to solve register allocation has proven challenging, resulting in only recent endeavors in this direction [17], [27], [42]. One of the primary challenges faced by researchers is the problem complexity and the scarcity of available training data. Since this area remains relatively unexplored in research, dedicated datasets for register allocation-related models are nonexistent. Consequently, studies in this domain have resorted to alternative methods for training their models, such as generating random graphs that do not represent real-world code [17], [27], or conducting training directly within the compiler's execution environment via reinforcement learning [42].

The recurrent issue of insufficient specialized datasets in the context of register allocation has also been observed beyond the scope of machine learning model research. Frequently, the scarcity of test cases poses a significant challenge in the development of novel allocators, as well as in the learning of register allocation techniques.

Therefore, to facilitate future research concerning the application of machine learning to register allocation and to address the scarcity of test cases for register allocation studies, this work proposes the development of an interference graph dataset. A graph interference generator from real C/C++ code, a tool for creating and expanding the dataset, is also developed and made available alongside the dataset. Although it is composed of PBQP graphs, the dataset is versatile, and widely tested in different contexts. This versatility allows its use with other register allocation techniques, such as graph coloring [13] and linear scan [37]. The dataset can also be used in educational contexts, providing real-world examples for the practical teaching of register allocation.

Our main goal is to accurately represent the codes as graphs and capture the potential costs of each allocation for training machine learning models, as high-quality data is essential for enabling machine learning algorithms to develop effective heuristics [18]. This initiative seeks to facilitate the creation of novel models and stimulate further exploration of this topic.

II. REGISTER ALLOCATION

Owing to their location in the central processing unit (CPU), registers are the fastest units in the memory hierarchy. They are often the only locations that can be directly accessed by most operations [16]. On the other hand, memory operations are energetically costly and inefficient compared with registers. A single memory access involves several instruction cycles, whereas two registers can be read, and one can be written, via only one instruction cycle [9].

Therefore, code variables should preferably be stored in registers. The number of registers is quite limited. For instance, the x86-64 chip contains 16 general purpose registers [4]. As a result, it is impossible to map all values to registers in most cases. In such cases, certain variables need to be stored in memory, with their values reloaded into registers just before use in a process called spill code generation [13].

Therefore, efficient register management methods are essential for optimizing variable mapping. The effectiveness of a register allocator has a direct impact on the performance of compiler-generated code.

A. GRAPH COLORING

The most common abstraction for register allocation is the interference graph. In this graph, vertices represent the live range of variables - the portions of code where a variable's value needs to be stored in a register. The edges represent interferences between these live ranges, with a vertex's degree indicating the number of edges connected to it. Consequently, register allocation is reduced to the graph coloring problem, where each color represents a physical register of the processor [13].

A graph is said to be k-colorable if it is possible to assign one of k colors to each vertex in such a way that no two connected vertices share the same color, which is the primary objective of register allocation. If the graph is not k-colorable, it becomes necessary to perform the spill process, which involves using memory to store variables whose live ranges are represented by the graph's vertices. When spills cannot be avoided, the goal shifts to minimizing their cost [13].

The first graph coloring register allocation algorithm was proposed by Chaitin et al. [13] and later adapted by Briggs et al. [10]. The algorithm begins with the removal of vertices from the graph, which are inserted into a stack. Removal always occurs with the vertex of the lowest degree; in the case of two vertices with the same degree, the vertex with the lower spill cost is removed. Then, the vertices at the top of the stack are re-added to the graph, which are already associated with a different color than their neighbors. If there are no available colors during insertion, the live range is marked for spill. The order of stack insertion aims to color

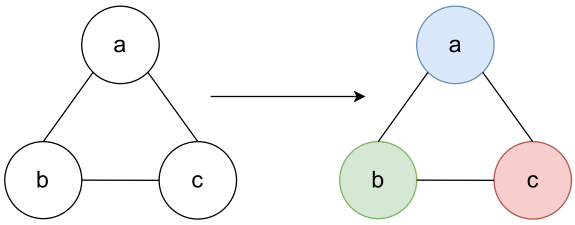


FIGURE 1. Example of graph coloring with 3 registers.

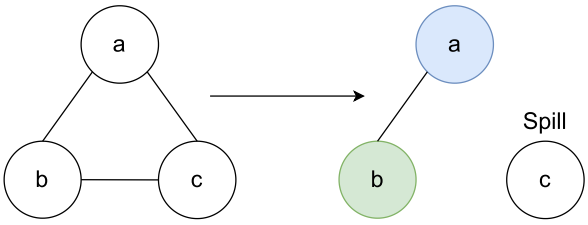


FIGURE 2. Example of graph coloring with 2 registers with spill.

as many vertices as possible and to preferably allocate live ranges with a higher spill cost [10], [13]. Figures 1 and 2 show two graph coloring examples with the same graph, with 3 and 2 colors respectively.

B. PBQP

The partitioned boolean quadratic problem (PBQP) is a specialized instance of the quadratic assignment problem [12]. It involves a scenario of multiple interdependent choices associated with costs. Efficient algorithms exist, particularly for sparse graphs, offering near-optimal solutions in linear time relative to the number of edges [39].

Graphs serve as the primary representation for this problem. In this representation, each node signifies a choice, with exactly one option to be selected. Each node is linked to an array containing the costs associated with its available choices. Edges symbolize dependencies between these choices and are linked to a cost matrix. An edge connecting a node with an array of dimensions n to another with an array of dimensions m corresponds to a matrix of dimensions $n \times m$ [12]. Figure 3 provides an illustrative example of graph coloring in the context of PBQP.

Selecting options in two interdependent nodes implicitly means selecting a cost value from the edge matrix. For instance, selecting option i from one node and option j from another corresponds to selecting the cost from row i and column j in the edge matrix. In the example depicted in Figure 3, the diagonals of the matrices contain infinite values, as neighboring nodes cannot share the same color. A solution is deemed valid if the sum of all the selected costs is finite. The primary objective of PBQP is to find the solution with the minimum cost [12].

By considering colors as registers and nodes as the live ranges of the variables, it is possible to abstract the register allocation problem to PBQP. The main advantage of this approach lies in its flexibility due to the cost associated with coloring decisions, which is advantageous for representing irregular architectures [27].

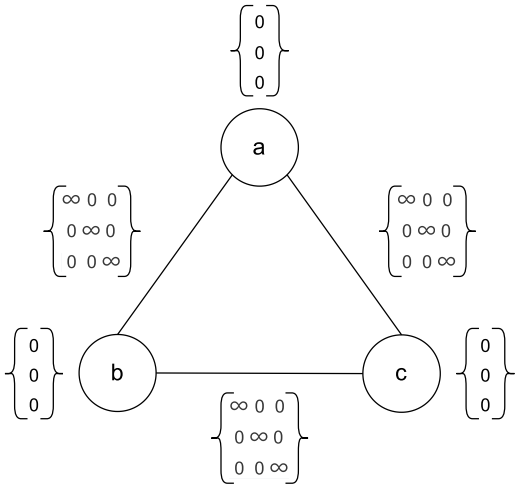


FIGURE 3. Example of PBQP graph coloring [12].

C. LINEAR SCAN

The linear scan algorithm offers a faster and less complex method for register allocation, resulting in code that approaches the efficiency of allocators via more intricate techniques, as outlined by Poletto and Sarkar [37]. Consequently, it has emerged as a compelling choice for applications where compilation time is a critical concern, notably in dynamic compilers and "just-in-time" compilers [37].

In the linear scan algorithm, the input comprises the number of registers and a list of live-ranges ordered in ascending order of their start times. The number of registers involved in interference only changes when a live-range ends or begins. Consequently, the algorithm transitions from one live-range's starting point to the next, allocating the current live range at the end of each iteration.

Within each iteration, a list is maintained that contains live ranges that interfere with the current live range and have been assigned to a register in past iterations. When a new starting point is encountered, the list is examined to eliminate "expired" live-ranges that no longer interfere with the new current live range. The size of this list remains within the bounds of the number of registers. If it is not possible to add a new life-range to the list, a live range is spilled. Figure 4 illustrates the live ranges of a simple code.

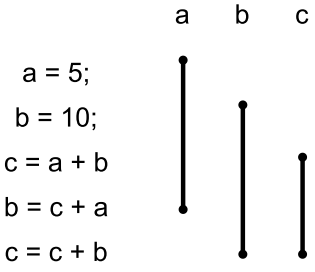


FIGURE 4. Example of live ranges in linear scan.

With 3 registers, the algorithm can easily allocate a register for each variable from the example. However, with 2 registers, spill is needed. In the third iteration, there are no registers available for the variable *c*. In this scenario, the linear scan chooses one of the other 2 variables for spill. The chosen variable is likely *a*, because it has fewer uses.

III. MACHINE LEARNING IN REGISTER ALLOCATION

With the advancement of machine learning techniques, and their increasing usage in heuristic development, several applications have been proposed across various computer science domains [6], [19], [25], [43]. In this context, its application in register allocation is an interesting alternative for generating new allocation approaches and, consequently, more optimized code. However, owing to the scarcity of available training data, the challenge of modeling neural networks to solve the register allocation problem, and the need to ensure the correctness of their solutions [42], the approaches to register allocation via machine learning have only recently been explored.

Among recent research efforts, Das et al. [17] employed deep learning to create new heuristics for graph coloring-based register allocation algorithms. To model the graph coloring problem, they utilized long short-term memory (LSTM) [24], a special type of recurrent neural network (RNN) [32]. An RNN is a class of neural network where connections between neurons can form a cycle, allowing it to retain information from past iterations of the neural network [32]. LSTMs maintain an internal record of the data (state), allowing them to retain information over longer periods than simpler RNNs [24]. Random graphs were employed for training the model. These graphs were generated via *very nauty* [8], a C library of graph algorithms, specifically designed for the rapid generation of random graphs.

Furthermore, VenkataKeerthy et al. [42] developed an application based on hierarchical multi-agent reinforcement learning [31] that handles the entire register allocation process. Register allocation is divided into four tasks, each assigned to an agent with varying levels of hierarchy. After taking its action, an agent calls the next lower-level agent. The agents are as follows:

- *Node selector*: Select a variable for allocation and determining the allocation order;
- *Task selector*: Decide whether the chosen variable is allocated to a register or undergoes the live range splitting process;
- *Splitter*: Split live ranges to minimize the spilling cost;
- *Coloring agent*: Choose a valid color for the variable, or perform a spill if no register is available.

The interference graphs are derived from the machine intermediate representation (MIR) of LLVM, a human-readable intermediate representation in the compilation phase [3]. These graphs are used as inputs to a gated graph neural network (GGNN) model [30], which learns to generate the state representation that feeds the node selector agents. The training was conducted using 2,000 randomly collected files from the SPEC CPU 2017 benchmarks and the Boost C++ library.

Moreover, Kim et al. [27] developed a deep reinforcement learning approach to address PBQP-based register allocation, which is specifically tailored for automated test equipment (ATE), a specialized embedded system for testing DRAM memory chips. The allocator was trained via randomly generated graphs produced by the Erdos-Rényi model [20].

This allocator uses Monte Carlo tree search (MCTS) [11], a heuristic and probability-based search technique that combines elements of classical tree search with reinforcement learning principles. Its primary use case is in software for playing board games.

Additionally, the deep reinforcement learning model from AlphaZero was also employed. The model was developed by the artificial intelligence research company DeepMind to master chess, shogi, and go games [40]. In this context, register allocation via PBQP is abstracted as a game and resolved via a Monte Carlo tree search algorithm, which uses AlphaZero's neural network to obtain the simulation results.

IV. MOTIVATION

One of the major challenges faced by researchers in this field is the lack of an appropriate dataset for training machine learning models applied to register allocation. To the best of our knowledge, ours is the first dataset tailored for these models. Consequently, researchers in the field have had to resort to alternative methods for training, such as reinforcement learning during compiler execution or generating random graphs. In this context, the rise of generative AI models could contribute to the generation of synthetic data for training in this field.

However, random graphs may not faithfully simulate the characteristics of different programs, thus limiting the learning potential of machine learning models. Moreover, another important factor to consider when training machine learning models for register allocation is the spill costs of the variables. The generation of random spill costs may not be consistent with values found in real code. Furthermore, the lack of training data is a problem even outside the domain of machine learning applications, serving as a hurdle in the learning of register allocation and the development of new allocators.

To solve this problem and assist future research, we created a dataset to address the lack of specialized training data for register allocation and to encourage future research on machine learning applied to this topic. Our dataset accurately represents real codes and their potential spill costs, enabling machine learning algorithms to develop new effective heuristics and approaches for register allocation.

Our dataset was designed to have versatility across multiple contexts. The graphs were generated from machine-independent intermediate representations and the dataset is easily adaptable to different register sets with varying constraints. This flexibility enables its utilization

across diverse architectures. Moreover, despite being initially in PBQP form, the data within the graphs can be effectively leveraged in conjunction with other register allocation techniques, such as linear scan, graph coloring, and other approaches. The dataset can also serve in educational settings, offering real-life examples for practical learning of register allocation and spill code minimization.

V. INTERFERENCE GRAPH GENERATOR

A. GRAPH GENERATION

To build the interference graph of a code, we analyze its intermediate compilation representation, extracting information about the live ranges and spill cost of each variable. This process was carried out with C/C++ files from various sources. To create the code's interference graphs, we developed a graph generation script, which is provided along with the dataset. This script allows the addition of interference graphs from other C/C++ files to the dataset. It also allows regenerating the graphs of the dataset with different parameters, such as the inclusion or customization of cost vectors and matrices. These script features are essential for ensuring the flexibility and adaptability of the dataset to different problems and contexts.

The graph generation process uses Python programs and LLVM tools. LLVM is a compiler framework designed to provide a flexible infrastructure for new compiler development and research [29]. The generated graphs are stored in files in JSON format. The graph generation process is shown in Figure 5.

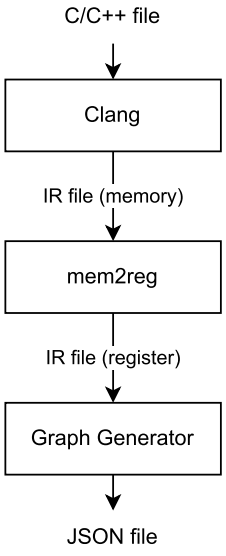


FIGURE 5. Graph generation sequence.

First, the codes were compiled with the clang front-end [1], resulting in LLVM intermediate representation (IR) files. The IR is a machine independent and human-readable low-level programming language, used as an intermediary stage within the LLVM compiler framework.

Typically, the IR represents the variables with memory references, which is not suitable for creating interference graphs. For this reason, we apply the transformation pass *mem2reg* to generate the intermediate representation code with virtual register references [2]. Therefore, the Python graph generator can traverse the IR files, identify the virtual registers in the code and gather data from each one, such as the type of stored value, the lines where it is used and the depth into the loop structures of each usage, among others. On the basis of this information, the program is able to generate interference graphs for each function in the source code.

To create the graphs, the script, initially, identifies the code of each function within the file, since each function is represented by a different graph. Then, the script extracts the virtual registers from the functions, which are always prefixed with a '%' character followed by an integer or a period. In this way, the program can also identify the number of uses of each virtual register. Additionally, in the first appearance of the virtual register, that is, in its definition, the type of the stored value is indicated in the code.

To identify the depth of each register usage, we take advantage of the SSA (static single assignment) structure of LLVM. SSA means that each virtual register receives only a single value assignment [12]. Owing to this characteristic, special handling is required in cases of branches and loops in the code, as an assignment may receive values from different registers depending on the preceding block of code. These assignments are called ϕ-nodes and have a special notation in the IR. Figure 6 shows a ϕ-node in a loop. From the ϕ-nodes, we can verify which blocks are inside a loop and, consequently, identify the depth of each usage of a virtual register.

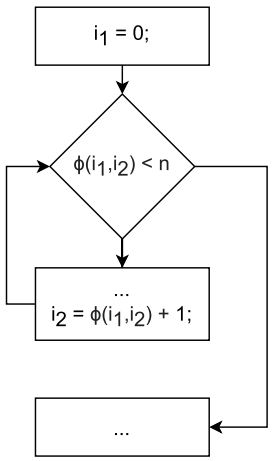


FIGURE 6. ϕ-node in a loop.

B. GRAPH STRUCTURE

The structure of the graph files was designed to facilitate their usability and understanding. Additionally, information related to the corresponding virtual register extracted from the source code is provided at the vertices. This information allows for the calculation of the main methods used to estimate the spill cost of the virtual register.

Each graph file contains an object in which the keys are the names of functions, and the values are their respective graphs. The graphs consist of JSON objects separated into vertices (*nodes*) and *edges*. The value related to the key nodes is an object containing all the vertices, each associated with the name of the virtual register linked to a variable from LLVM intermediate code representation. The vertices have the following keys and values:

- *type*: indicates the type of value stored in the register. The types are:
  - *i8-64*: integer values classified by their size, which can be 8, 16, 32 and 64 bits;
  - *float*: floating point number with 32 bits;
  - *double*: double floating point number with 64 bits;
  - *ptr*: pointer to an address or label;
  - *unidentified*: when the type is unidentified.
- *uses*: an array with a size equal to the number of uses of the virtual register. Each value indicates an instruction number in the IR;
- *uses deepness*: an array with a dimension equal to the size of ''uses''. The values in the array refer to the depth in the loop structures of each virtual register usage. For example, if a value is not inside any loop, its depth value will be zero. If it is inside one loop, its depth will be one; if it is inside two nested loops, it will be two, and so on;
- *cost array*: an array of costs for each allocation option. It provides the costs of associating the virtual register with other virtual registers, along with the cost of spilling the value. The graph generation program offers an option to omit the cost array from the output file, generating simple interference graphs. By default, the array contains 8 registers with an allocation cost of 0.

The value related to the key *edges* is an array of edges. Each edge contains attributes *node 1* and *node 2*, which indicate the interference represented by the edge. Note that the order of vertices does not matter. Moreover, the edges also include the cost matrix as a third attribute. Similar to the cost array, it is possible to create graphs without the cost matrix at the edges. Figure 7 shows an example of a graph represented in JSON format.

The main objective of this graph organization is to maintain dataset versatility, enabling its utilization with various register allocation techniques. PBQP graphs can be easily employed as basic interference graphs. In this context, the graph generator offers an option to omit the cost array and cost matrix from the graph, effectively creating a non-PBQP interference graph. Moreover, the *uses* attribute allows for the definition of live ranges for each virtual register for use in linear scan algorithms [37].

Additionally, users have the flexibility to create spill cost functions on the basis of various attributes, such as *type*, *uses* and *uses deepness*. This capability enables the application of context-specific heuristics in conjunction with the dataset, or even facilitates tests and the development of novel methods for calculating variable spill costs.

```
"@main" :   {
    "nodes" : {
        "%1" : {
            "type" : "i8",
            "uses" : [1,2,3,4,6],
            "uses deepness" : [1,1,2,3,4],
            "cost array" : [0,0,0,9]
        },
        "%2" : {
            "type" : "ptr",
            "uses" : [5,6,8,11],
            "uses deepness" : [1,2,2,1],
            "cost array" : [0,0,0,6]
        }
    },
    "edges" : [
        {
            "node 1" : "%1",
            "node 2" : "%2",
            "cost matrix" : [["Infinity",0,0,9],
                             [0,"Infinity",0,9],
                             [0,0,"Infinity",9],
                             [6,6,6,15]]
        }
    ]
}
```

FIGURE 7. A function graph example in a JSON output file.

Furthermore, the graph generator offers additional options to tailor the dataset according to the user's needs. By default, an output file can contain multiple graphs, one for each function in the input file, but it is possible to specify that each generated graph should be stored in a separate output file. As depicted in Figure 7, the cost array and cost matrix have generic values by default. The main reason is to keep the graph architecture-independent. Therefore, the user can specify the number of registers and their weights as needed, which allows the simulation of various processor architectures. Finally, the code generator allows the user to add new graphs to the dataset on the basis of their own code samples.

C. GRAPH EXAMPLE

In this subsection, we present a simple example of a code transformation into a graph in the dataset. Figure 8 shows a simple C code example, which generates the graph presented in Figure 9.

```
void main(){
    int m[10][10];
    int n = 10;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            m[i][j] = 0;
        }
    }
    r = m[0][0];
}
```

FIGURE 8. A simple C code example.

Finally, Figure 10 presents the interference graph in JSON format used in the dataset. Importantly, this is not the resulting graph from the graph generation process employed

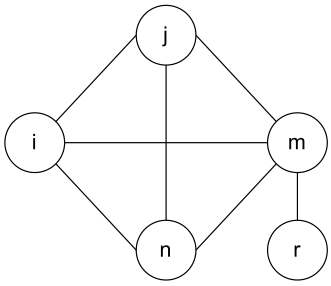


FIGURE 9. The C code interference graph.

in the dataset. To keep the example straightforward, the intermediate representation step was omitted. As a result, the interference graph was constructed directly from the code's variables rather than from the virtual registers. Furthermore, in an effort to maintain the simplicity of the example, the option to exclude cost vectors and matrices was also utilized.

VI. DATASET ANALYSIS

A. DATASET STRUCTURE

The dataset is organized according to the source of the code samples. It is divided into three subsets: *Basic*, *CodeNet* and *Spec*. Table 1 shows the number of files and the number of interference graphs of functions in the entire dataset, and in each subset.

TABLE 1. Number of files and graphs on the dataset and subsets.

Subset	Number of files	Number of graphs
Total	5889	24688
Basic	1555	3549
CodeNet	3939	18939
Spec	395	2200

The *Basic* subset contains the code graphs extracted from open-source GitHub repositories. It primarily consists of C/C++ code from programming logic exercises and programming course examples.

The *CodeNet* subset contains graphs of code samples from the CodeNet dataset [38]. CodeNet is a large-scale dataset with approximately 14 million code samples, each of which is an intended solution to one of approximately 4000 coding problems. The code samples are obtained from submissions from the online judge websites like AIZU Online Judge and AtCoder. The code samples are written in more than 50 programming languages, especially C++, C, Python, and Java [38]. Only one code submission per problem was selected. This decision was made because, despite being submissions from different users, solutions to the same problem often share very similar code structures. Consequently, to avoid adding redundant interference graphs to the dataset, a random C or C++ submission was chosen for each problem.

Finally, the *Spec* subset comprises code extracted from the SPEC CPU 2017 benchmarks [5]. Only a few code files from the benchmarks were selected because many files are


```
"main" :   {
    "nodes" : {
        "n" : {
            "type" : "i8",
            "uses" : [2,3,4],
            "uses deepness" : [1,2,3]
        },
        "i" : {
            "type" : "i8",
            "uses" : [3,5],
            "uses deepness" : [2,3]
        },
        "j" : {
            "type" : "i8",
            "uses" : [4,5],
            "uses deepness" : [3,3]
        },
        "m" : {
            "type" : "ptr",
            "uses" : [1,5,6],
            "uses deepness" : [1,3,1]
        },
        "r" : {
            "type" : "i8",
            "uses" : [6],
            "uses deepness" : [1]
        }
    },
    "edges" : [
        {
            "node 1" : "n",
            "node 2" : "m"
        },
        {
            "node 1" : "n",
            "node 2" : "i"
        },
        {
            "node 1" : "n",
            "node 2" : "j"
        },
        {
            "node 1" : "i",
            "node 2" : "m"
        },
        {
            "node 1" : "i",
            "node 2" : "j"
        },
        {
            "node 1" : "j",
            "node 2" : "m"
        },
        {
            "node 1" : "m",
            "node 2" : "r"
        }
    ]
}
```


FIGURE 10. A function graph example in a JSON output file.

incompatible with the graph generation process employed. Many files consisted of definitions or had dependencies that made them compilable only during the benchmark build. For this reason, only files that could be compiled individually were selected. Unlike the other subsets, *Spec* contains subdivisions on the basis of the source benchmarks of the code. Table 2 presents the subdivisions (benchmarks) of the subset along with their respective numbers of files and graphs.

TABLE 2. Number of JSON files and graphs on each *Spec* benchmark subset.

Benchmark	Number of files	Number of graphs
500.perlbench_r	6	28
502.gcc_r	29	547
505.mcf_r	20	58
507.cactuBSSN_r	8	10
511.povray_r	1	3
519.lbm_r	2	21
521.wrf_r	15	86
526.blender_r	177	581
527.cam4_r	6	6
531.deepsjeng_r	38	216
541.leela_r	8	26
544.nab_r	13	153
557.xz_r	24	28
600.perlbench_s	3	14
602.gcc_s	12	248
605.mcf_s	10	29
628.pop2_s	20	124
999.specrand_ir	2	22

Additionally, the dataset includes the IR files that were used to generate the interference graphs. These files are organized in the same manner as the graph subsets. The purpose of retaining the IR files alongside the graphs in the dataset is to allow the graphs to be recreated via different generation options, as mentioned in Section V-B.

B. DATASET STATISTICS

In this subsection, we present dataset statistics, illustrating the distributions of the numbers of nodes, edges, and node degrees in the graphs. Table 3 displays the quantity of graphs throughout the dataset and their subsets, categorized by the number of vertices falling within specific intervals.

TABLE 3. Number of graphs for node quantity on the dataset and subsets.

Number of nodes	Basic	CodeNet	Spec	Total
0 - 15	1749	9385	897	12031
16 - 31	804	3015	397	4216
32 - 47	403	1464	215	2082
48 - 63	188	1233	118	1539
64 - 79	120	717	106	943
80 - 95	81	557	78	716
96 - 111	50	457	61	568
112 - 127	50	328	35	413
$\geq$ 128	104	1783	293	2180

Table 4 displays the quantity of graphs throughout the dataset and their subsets, categorized by the number of edges falling within specific intervals.

TABLE 4. Number of graphs for edge quantity on the dataset and subsets.

Number of nodes	Basic	CodeNet	Spec	Total
0 - 15	1139	5941	489	7569
16 - 47	571	3064	364	3999
48 - 79	330	1469	196	1995
80 - 127	288	1118	158	1564
128 - 191	246	950	111	1307
192 - 255	193	623	99	915
256 - 511	345	1600	178	2123
512 - 1023	214	1530	215	1959
$\geq$ 1024	223	2644	390	3257

Table 5 displays the quantity of nodes throughout the graphs from the dataset and their subsets, categorized by the node degrees falling within specific intervals.

TABLE 5. Number of nodes for nodes degree on the dataset and subsets.

Number of nodes	Basic	CodeNet	Spec	Total
0 - 2	2127	10915	768	13810
3 - 5	813	5122	779	6714
6 - 8	381	1814	429	2624
9 - 11	138	672	96	906
12 - 14	55	238	48	341
15 - 17	10	88	33	131
18 - 20	9	59	8	76
21 - 23	4	19	3	26
$\geq$ 24	12	12	36	60

Figures 11, 12, and 13 depict histograms of the dataset's overall distributions for the number of nodes, number of edges, and node degrees, respectively.

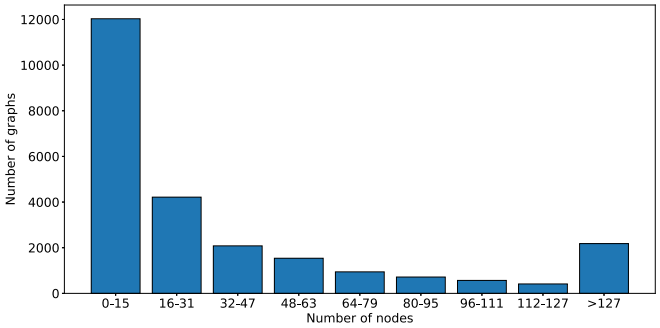


FIGURE 11. Number of graphs for the node quantity on the dataset.

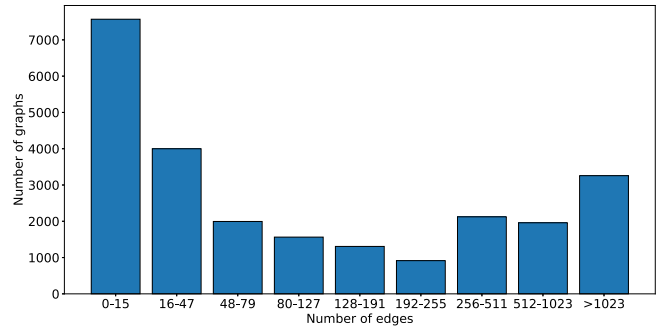


FIGURE 12. Number of graphs for the edge quantity on the dataset.

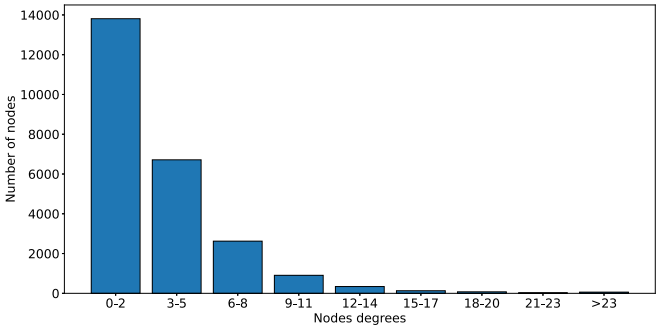


FIGURE 13. Number of nodes for the node degree on the dataset.

On the basis of Tables 3 through 5 and Figures 11 to 13, we concluded that, in general, the graphs in the dataset are denser rather than sparse. Furthermore, we observed that the dataset contains a significant number of graphs generated from small functions.

We can also observe that, although it is the minority, there is a considerable number of nodes with a high degree. Nodes with a degree less than the number of available registers are always allocated. Thus, nodes with higher degrees are necessary to create complex allocation scenarios, especially considering that the more interference a node has, the greater its chance of experiencing a spill. In this context, the number of high-degree nodes may seem low; however, in real-world codes, the majority of variables have a low degree [9]. Therefore, it is important to maintain a distribution consistent with real-world codes to provide better learning for machine learning algorithms.

VII. USE CASE EXAMPLES

To demonstrate potential research applications of the dataset, in this section, we present examples of register allocation algorithms using the dataset. We present three algorithms for register allocation:

- *Linear Scan*: A simple, well-established and widely used register allocation algorithm. This example demonstrates that the dataset can be used alongside traditional register allocation techniques. Additionally, this example highlights the adaptability of the dataset, which can be used by algorithms even if they do not directly employ interference graphs;
- *Genetic algorithm*: A register allocation algorithm based on a genetic algorithm. This example illustrates the flexibility of the dataset, showing the possibility of using the dataset for new allocation approaches that differ from commonly used algorithms. Additionally, this example highlights the potential to adapt the dataset to unconventional input formats.
- *LSTM model*: An LSTM neural network model recently proposed by Das et al. [17]. This example demonstrates the use of the dataset for training machine learning models applied for register allocation.

The results of the use cases are presented considering the Spec subset, as the remaining dataset was utilized as training data for the LSTM model.

A. LINEAR SCAN

To showcase the dataset's utility alongside traditional register allocation approaches, we employed it in conjunction with Linear Scan. As outlined in subsection II-C, Linear Scan stands out as one of the most straightforward and widely employed methodologies for register allocation. Employing the dataset as input for this algorithm necessitates the creation of live ranges derived from the dataset graphs. Consequently, live ranges were established for each node, utilizing the initial and final values of the *uses* attribute associated with the nodes.

In instances where spill becomes necessary, the variable with the minimal spill cost within the actives live ranges in that iteration is chosen. The spill cost of each node is calculated via the spill cost heuristic of Chaitin et al. [13], as presented in Equation 1.

$$cost = \sum_{I \in \text{uses deepness}} 10^I \tag{1}$$

Figures 14 and 15 show the results of applying linear scan to the *Spec* subset. Figure 14 depicts the distribution of the percentages of allocated nodes throughout the graphs. Similarly, Figure 15 presents the distribution of spill cost percentages. The spill cost percentage is calculated in relation to the sum of the spill costs of all the variables in the graph.

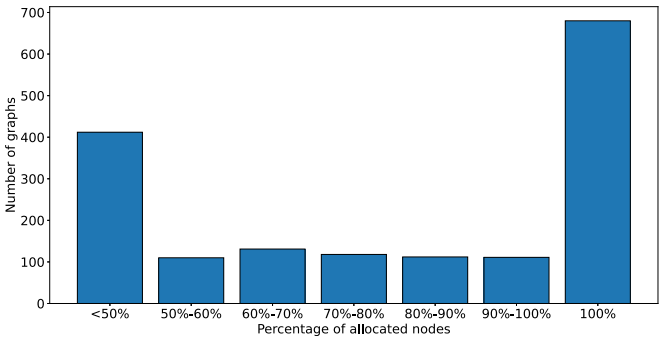


FIGURE 14. Distribution of graphs by allocated nodes percentage.

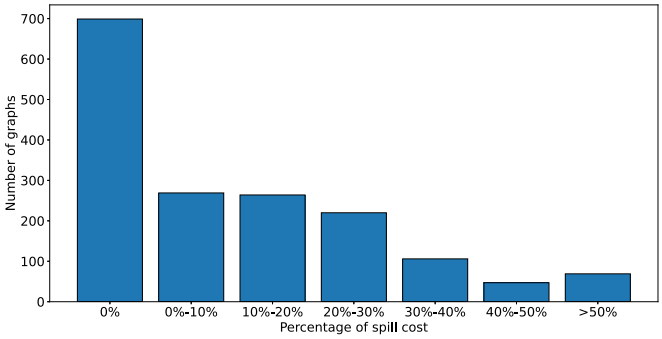


FIGURE 15. Distribution of graphs by spill cost percentage.

B. GENETIC ALGORITHM

To demonstrate the versatility of the dataset, we implemented a register allocation model based on a genetic algorithm, which illustrates that the dataset can be used with innovative register allocation proposals. A genetic algorithm (GA) is an optimization algorithm that is inspired by natural selection. It is a population-based search algorithm, that uses the concept of survival of the fittest [33]. Figure 16 shows the flowchart of the genetic algorithm applied in this use case.

The genetic algorithm starts by generating a random population, where each population comprises a set of chromosomes, with each chromosome representing a potential

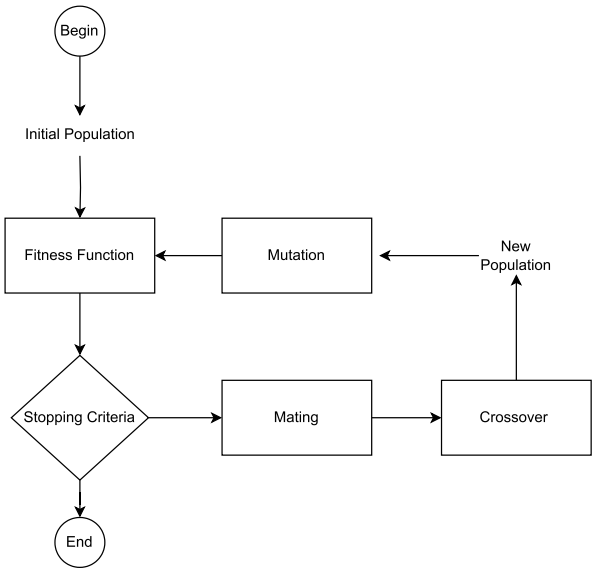


FIGURE 16. Flowchart of the genetic algorithm.

solution to the problem. Furthermore, chromosomes are composed of genes, which are values that encapsulate the characteristics of the solutions. The selection step subsequently takes place, wherein the fittest individuals (chromosomes), those with more suitable solutions, are chosen to form the new population. This selection is performed via the fitness function, which returns a value between 0 and 1 to denote the quality of the solution [26].

From the selected chromosomes, new individuals are created to constitute the new population. The generation of new chromosomes is called crossover and involves combining different genes from two "parent" individuals, as illustrated in Figure 17.

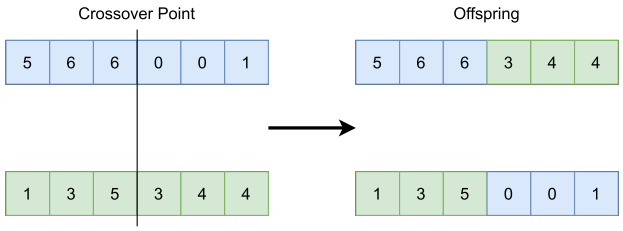


FIGURE 17. Creation of new chromosomes through crossover.

Moreover, before proceeding to the next iteration, the mutation stage occurs, entailing a small chance of altering a gene to another random value [26].

For register allocation, chromosomes represent a potential allocation. Therefore, a chromosome consists of n genes, where each corresponds to a variable. Considering the utilization of the dataset, each gene corresponds to a node in the graph. The gene value represents a potential allocation. With m registers available for allocation, a gene can take a value between 0 and m, where values from 0 to m - 1 are associated with a register each, and m is associated with spill. Before the selection process, it is necessary to ensure that the allocations in the chromosomes are valid. Thus, the edges of the graphs are traversed to check for interference. Invalid allocations are replaced by spills. Additionally, the fitness function is calculated according to Equation 2.

$$fitness = 1 - \frac{spill\ costs}{total\ spill\ cost} \tag{2}$$

The spill cost value consists of the sum of the spill costs that occurred in the solution, which is calculated in the same manner as in Linear Scan, as shown in Equation 1. The total spill cost corresponds to the sum of the spill costs of all the variables present in the interference graph. Thus, the lower the spill cost is, the better the fitness function value, with a value of 1 corresponding to an allocation without spill.

Furthermore, for the application of the genetic algorithm, an allocation of 8 registers was considered, employing a population with 50 individuals, selecting 8 chromosomes for mating, and a mutation rate of 0.5%. A total of 40 generations (iterations) were used. Additionally, a stopping criterion was incorporated if an optimal solution was found.

Figures 18, 19 and 20 present the results of the genetic algorithm register allocation applied to the *Spec* subset. Figure 18 depicts the distribution of the percentages of allocated nodes throughout the graphs. Similarly, Figure 19 shows the distribution of spill cost percentages. Finally, Figure 20 presents the distribution of fitness function values throughout the graphs. Note that the fitness function is equivalent to the inverse of the spill cost percentage.

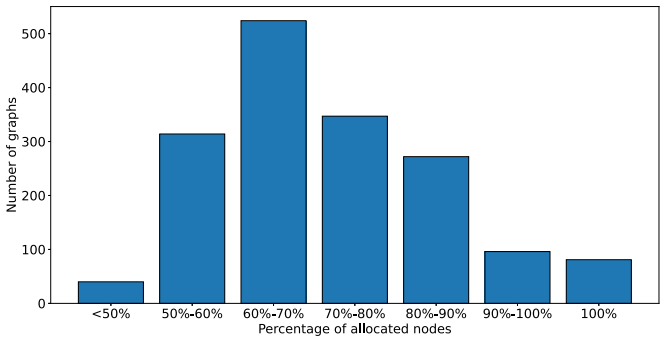


FIGURE 18. Distribution of graphs by allocated nodes percentage.

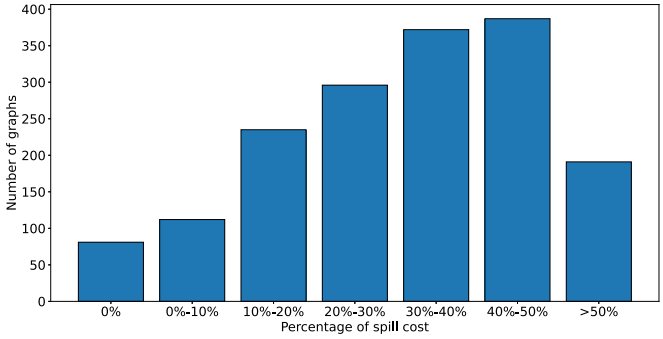


FIGURE 19. Distribution of graphs by spill cost percentage.

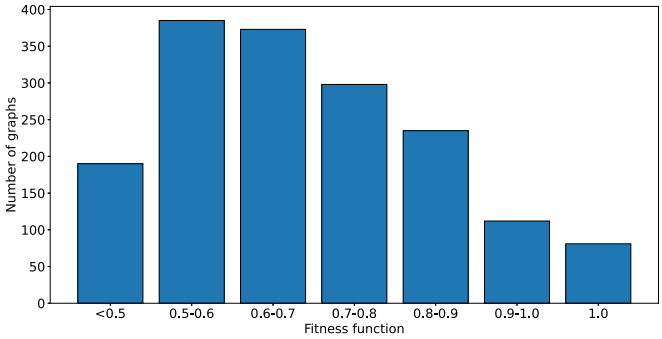


FIGURE 20. Distribution of graphs by function fitness value.

C. LSTM MODEL

To demonstrate the dataset's usability in machine learning-based register allocation approaches, we applied it in conjunction with one of the models recently proposed in the literature. As discussed in subsection III, this model was proposed by Das et al. [17], and employed LSTM neural networks for register allocation. This model has a slightly different objective than traditional register allocation, as it does not allow for spills. Thus, the neural network uses as many registers as needed to allocate the graphs, always aiming to utilize the smallest number of registers possible. Instead of trying to efficiently manage a finite number of registers, as typically pursued by allocators, this model aims to predict the minimum number of registers required to allocate the entire graph.

Das et al. [17] established a limit for the size of the interference graph, considering graphs with 100 vertices or fewer. Consequently, the entire adjacency matrix of the interference graph is used as input to the neural network. The input sequence for the LSTM consists of the sequence of graph vertices, and the output of the LSTM model is another list, where each index is associated with a register. Therefore, the index associated with the highest output value indicates the register allocated to the node. Thus, at each time step, the neural network receives an adjacency list of a node, and returns the integer number associated with the register assigned to the node, as illustrated in Figure 21.

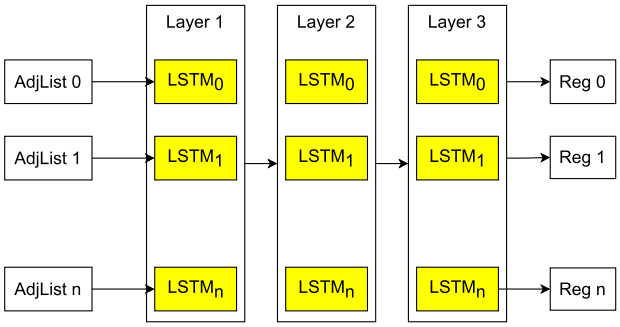


FIGURE 21. Diagram of the LSTM register allocation model.

For model training, graphs with 100 nodes or fewer from the *Basic* and *Codenet* subsets were utilized. Graph labels, representing the minimum number of registers required for full graph allocation, were obtained through brute force. Additionally, the graphs were processed to conform to the neural network input, thus forming the adjacency lists for each variable, as depicted in Figure 22.

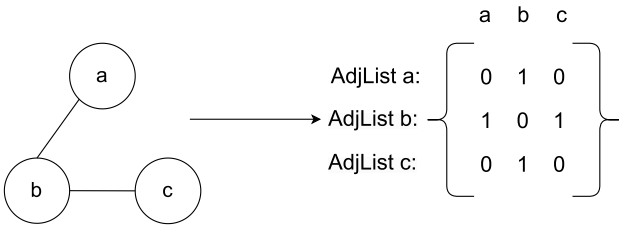


FIGURE 22. Example of the graph transformation to input to the LSTM model.

Furthermore, as pointed out by Das et al. [17], it is important to consider that the predictions generated by the neural network do not necessarily adhere to the interferences present in the graphs. Thus, the results obtained by the model may contain invalid allocations, potentially resulting in cases where the number of registers used is less than the minimum quantity specified in the graph labels. Therefore, it is necessary to apply a correction step to the predictions of the neural network to validate the allocations. During correction, nodes with invalid results receive new allocations. Thus, the nodes are assigned the smallest numbers associated with a free register.

Figures 23 and 24 present the results of the LSTM register allocation model applied to the *Spec* subset. Figure 23 depicts the average number of registers required for the allocation of graphs of the same size. The means of the predicted number of registers from graphs with the same number of nodes, both before and after the correction step, are presented, as well as are the means of the optimal number of registers for graphs of the same size. Figure 24 shows the same means from Figure 23 but now shifted relative to the averages of the optimal results, demonstrating the distance between the predictions obtained by the model and the best possible results.

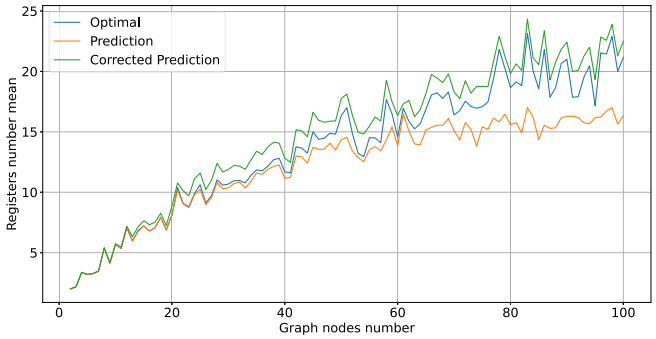


FIGURE 23. Number of register means by graph size.

We can observe in Figures 23 and 24 the importance of the correction step, as the predictions before this operation are

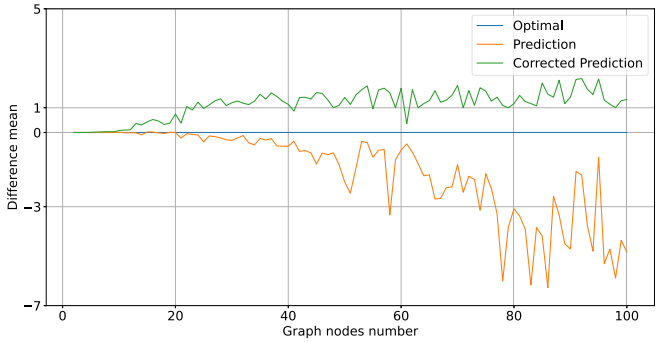


FIGURE 24. Difference mean from the optimal register number by graph size.

supposedly worse than the optimal results, meaning that they are invalid results. Additionally, it is possible to see that the corrected results are close to the optimal results, using fewer than 3 additional registers in most cases.

VIII. CONCLUSION

In this work, we introduced a dataset of interference graphs designed for training machine learning models focused on register allocation. We also presented a graph generator, enabling the expansion and adaptation of the dataset according to the demand in this research field.

The structure of the graphs in the dataset is intuitive and allows the storage of diverse data about the variables to be allocated. Consequently, their data can be easily processed and transformed to suit the allocator's requirements. Therefore, the dataset is versatile, permitting its use for different techniques and abstractions of register allocation. Moreover, its adaptability extends to its collaborative use with neural networks, enabling its use with new machine learning-based register allocation techniques.

With these features in mind, we anticipate that our dataset facilitates the development of new research on the subject. We expect our dataset to lead to the creation of novel solutions for the register allocation problem via machine learning, thereby contributing to improvements in the performance of compiler-generated code. We also hope that the dataset can contribute to educational contexts by providing practical examples for teaching register allocation, allowing students to observe the results of algorithms in various cases.

In future works we intend to expand the dataset by adding graphs representing the interference of new codes from different sources. One possibility worth exploring is the use of generative AI models to obtain code examples for generating new graphs. This approach ensures that the graphs reflect real codes, as opposed to merely generating random graphs directly. Additionally, we observed that by making the graphs generalized enough for use with different sets of registers, it can sometimes be challenging to emulate specific architectures. In the future, we plan to add tools to make the graphs more closely align with the assembly of specific architectures.

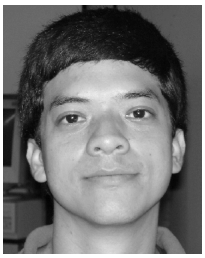
REFERENCES

[1] *Clang: A C language family frontend for llvm*. Accessed: Aug. 25, 2023. [Online]. Available: https://clang.llvm.org/
[2] *The Llvm Compiler Infrastructure*. Accessed: Aug. 25, 2023. [Online]. Available: https://llvm.org
[3] *Machine Ir (mir) Format Reference Manual*. Accessed: Mar. 27, 2023. [Online]. Available: https://llvm.org/docs/MIRLangRef.html
[4] *X64 Architecture*. Accessed: Oct. 26, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture?redirectedfrom=MSDN
[5] (2017). *SPEC CPU 2017 Benchmark Suite*. Accessed: Sep. 1, 2023. [Online]. Available: https://www.spec.org/cpu2017
[6] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surveys*, vol. 51, no. 5, pp. 1–42, Sep. 2018.
[7] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the NP-completeness proof of chaitin et al. Really prove? Or revisiting register allocation: Why and how," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.* Cham, Switzerland: Springer, 2006, pp. 283–298.
[8] K. Briggs. *The Very_nauty Graph Library*. Accessed: Mar. 24, 2023. [Online]. Available: http://keithbriggs.info/very_nauty.html
[9] P. Briggs, "Register allocation via graph coloring," Ph.D. thesis, Rice Univ., Houston, TX, USA, 1992. [Online]. Available: https://hdl.handle.net/1911/16537
[10] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, May 1994.
[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.
[12] S. Buchwald, A. Zwinkau, and T. Bersch, "SSA-based register allocation with PBQP," in *Compiler Construction*, J. Knoop, Ed., Berlin, Germany: Springer, 2011, pp. 42–61.
[13] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Lang.*, vol. 6, no. 1, pp. 47–57, Jan. 1981.
[14] W.-Y. Chen, G.-Y. Lueh, P. Ashar, K. Chen, and B. Cheng, "Register allocation for Intel processor graphics," in *Proc. Int. Symp. Code Gener. Optim.*, vol. 35, New York, NY, USA. ACM, Feb. 2018, pp. 352–364.
[15] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 4, pp. 501–536, Oct. 1990.
[16] K. Cooper and L. Torczon, *Engineering a Compiler: International Students Edition*. San Mateo, CA, USA: Morgan Kaufmann, 2003.
[17] D. Das, S. A. Ahmad, and V. Kumar, "Deep learning-based approximate graph-coloring algorithm for register allocation," in *Proc. IEEE/ACM 6th Workshop LLVM Compiler Infrastructure HPC (LLVM-HPC) Workshop Hierarchical Parallelism Exascale Comput. (HiPar)*, Nov. 2020, pp. 23–32.
[18] X. L. Dong and T. Rekatsinas, "Data integration and machine learning: A natural synergy," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA: ACM, May 2018, pp. 1645–1650.
[19] I. El Naqa and M. J. Murphy, "What is machine learning?" in *Machine Learning in Radiation Oncology: Theory and Applicationso*, I. El Naqa, R. Li, and M. J. Murphy, Eds., Cham, Switzerland: Springer, 2015, pp. 3–11.
[20] P. L. Erdos and A. Rényi, "On the evolution of random graphs," *Trans. Amer. Math. Soc.*, vol. 5, no. 1, pp. 17–60, 1960.
[21] X. Han, Z. Li, and Y. Xu, "Quantum assisted stochastic economic dispatch for renewables rich power systems," 2024, *arXiv:2404.13073*.
[22] X. Han and Y. Zhang, "Decomposition-coordination-based voltage control for high photovoltaic-penetrated distribution networks under cloud-edge collaborative architecture," *Int. Trans. Electr. Energy Syst.*, vol. 2022, pp. 1–20, Jan. 2022.
[23] J. Hartmanis, "Computers and intractability: A guide to the theory of NP-completeness (M. R. Garey and D. S. Johnson)," *SIAM Rev.*, vol. 24, no. 1, pp. 90–91, Jan. 1982.
[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[25] M. Karimi-Mamaghan, M. Mohammadi, P. Meyer, A. M. Karimi-Mamaghan, and E.-G. Talbi, "Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art," *Eur. J. Oper. Res.*, vol. 296, no. 2, pp. 393–422, Jan. 2022.
[26] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, Feb. 2021.
[27] M. Kim, J.-K. Park, and S.-M. Moon, "Solving PBQP-based register allocation using deep reinforcement learning," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2022, pp. 1–12.
[28] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, "Automatic construction of inlining heuristics using machine learning," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2013, pp. 1–12.
[29] C. Lattner and V. Adve, "The llvm compiler framework and infrastructure tutorial," in *Proc. Languages Compil. High Perform. Comput., 17th Int. Workshop, (LCPC)*, West Lafayette, IN, USA. Cham, Switzerland: Springer, 2004, pp. 15–16.
[30] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2017, *arXiv:1511.054932*.
[31] R. Makar, S. Mahadevan, and M. Ghavamzadeh, "Hierarchical multi-agent reinforcement learning," in *Proc. 5th Int. Conf. Auto. agents*, New York, NY, USA. ACM, May 2001, pp. 246–253.
[32] L. R. Medsker and L. Jain, "Recurrent neural networks," in *Design and Applications*. Boca Raton, FL, USA: CRC Press, 2001, pp. 64–67.
[33] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin, Germany: Springer, 1992.
[34] R. Mosaner, "Machine learning to ease understanding of data driven compiler optimizations," in *Proc. Companion ACM SIGPLAN Int. Conf. Syst., Program., Lang., Appl., Softw. Humanity*, New York, NY, USA: ACM, Nov. 2020, pp. 4–6.
[35] A.-C. Orgerie, M. D. D. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 1–31, Mar. 2014.
[36] F. M. Q. Pereira, "A survey on register allocation," Dept. Comput. Sci., Univ. California, Los Angeles, CA, USA, Tech. Rep., 2008.
[37] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.
[38] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," 2021, *arXiv:2105.12655*.
[39] B. Scholz and E. Eckstein, "Register allocation for irregular architectures," *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 139–148, Jun. 2002.
[40] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017, *arXiv:1712.01815*.
[41] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 77–90, May 2003.
[42] S. VenkataKeerthy, S. Jain, A. Kundu, R. Aggarwal, A. Cohen, and R. Upadrasta, "RL4ReAl: Reinforcement learning for register allocation," 2022, *arXiv:2204.02013*.
[43] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.
[44] T. Yang, X. Han, H. Li, W. Li, and A. Y. Zomaya, "Parallel scientific power calculations in cloud data center based on decomposition-coordination directed acyclic graph," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2491–2502, Mar. 2022.

PEDRO ZAFFALON DA SILVA received the B.S. degree in computer science from the State University of Londrina (UEL), in 2023, where he is currently pursuing the M.S. degree in computer science. Currently, he studies the application of neural network models in register allocation optimization.

HELEN CRISTINA DE MATTOS SENEFONTE received the M.S. degree in electronic and computer engineering from the Aeronautics Institute of Technology (ITA), in 2009, and the Ph.D. degree in electrical and computer engineering from the Federal University of Technology-Paran (UTFPR), in 2022. She is currently working as a Professor with the Department of Computer Science, State University of Londrina (UEL), Brazil, where her current research explores generative artificial intelligence (GenAI) and machine learning (ML) models. She has substantial experience in industry and academia and has been working with machine learning, since 2006.



WESLEY ATTROT received the Ph.D. degree in computer science from the University of Campinas (UNICAMP), in 2009. He is currently working as a Professor with the Department of Computing, State University of Londrina (UEL), Brazil, where his current research explores compiler optimization techniques and register allocation and spill code generation. He has substantial experience in academia, working with compiler construction and optimization, since 2002.