Universidade Estadual de Londrina

MATHEUS PIRES VILA REAL

APLICAÇÕES DO APRENDIZADO DE MÁQUINA NA GERAÇÃO DE CÓDIGO *SPILL*

LONDRINA

2023

MATHEUS PIRES VILA REAL

APLICAÇÕES DO APRENDIZADO DE MÁQUINA NA GERAÇÃO DE CÓDIGO *SPILL*

Versão Preliminar de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

LONDRINA

2023

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

Sobrenome, Nome.
Título do Trabalho : Subtitulo do Trabalho / Nome Sobrenome. - Londrina, 2017.
100 f. : il.

Orientador: Nome do Orientador Sobrenome do Orientador.
Coorientador: Nome Coorientador Sobrenome Coorientador.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2017.
Inclui bibliografia.

1. Assunto 1 - Tese. 2. Assunto 2 - Tese. 3. Assunto 3 - Tese. 4. Assunto 4 - Tese. I. Sobrenome do Orientador, Nome do Orientador. II. Sobrenome Coorientador, Nome Coorientador. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

MATHEUS PIRES VILA REAL

APLICAÇÕES DO APRENDIZADO DE MÁQUINA NA GERAÇÃO DE CÓDIGO *SPILL*

Versão Preliminar de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

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

Prof. Dr. Segundo Membro da Banca
Universidade Estadual de Londrina

Prof. Dr. Terceiro Membro da Banca
Universidade Estadual de Londrina

Londrina, 24 de novembro de 2023.

AGRADECIMENTOS

"Não vos amoldeis às estruturas deste mundo, mas transformai-vos pela renovação da mente, a fim de distinguir qual é a vontade de Deus: o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2))

REAL, M. **Aplicações do Aprendizado de Máquina na geração de código *spill***. 2023. 73f. Trabalho de Conclusão de Curso – Versão Preliminar (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

A alocação de registradores é uma das otimizações de código mais significativas do processo de compilação de um programa. Mas, devido à natureza dos algoritmos empregados nas implementações tradicionais, ela caracteriza-se como um prolema NP-completo, o qual é difícil de se resolver de maneira ótima. Nesse contexto, ao longo dos anos foram sendo propostas heurísticas e ajustes visando tornar as técnicas de geração de código *spill* mais precisas e menos custosas. Com o crescimento da relevância do aprendizado de máquina (*machine learning*) — área da inteligência artificial, que engloba o desenvolvimento de modelos complexos de categorização e predição treinados de maneira algorítmica — surge a perspectiva de integração de ambas as áreas. Isto posto, este trabalho se propõe a vislumbrar o estado da arte da aplicação do aprendizado de máquina na alocação de registradores e investigar as possibilidades práticas de implementação de alocadores utilizando modelos treinados por *machine learning*.

Palavras-chave: Compiladores. Aprendizado de Máquina. Código spill. Alocação de registradores. Otimização

REAL, M. **Machine Learning applications in spill code generation**. 2023. 73p. Final Project – Draft Version (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

Register allocation is highlighted as one of the key code optimizations throughout a program's compilation process. It is, however, a NP-complete problem, due to the nature of the traditional implementations, which are unable to obtain optimal solutions in reasonable time. In this context, several heuristics and adjustments have been proposed over the course of the years in order to improve the precision and the speed of register allocators. Machine learning, in the other hand, is a field of artificial intelligence encompassing the development of complex classification and prediction models which are trained in an algorithmic way, that grew in relevance and became widely researched recently. Therefore, efforts to apply machine learning techniques to register allocation, aiming to improve spill code generation, arose. This work is intended to examine the current state-of-the-art of the integration of both fields and investigate the implementation possiblities of register allocators that widely employ machine learning models in their inner workings.

Keywords: Compilers. Machine Learning. Spill code. Register allocation. Optimization.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	*Central processing unit*
GPU	*Graphics processing unit*
CFG	*Control-flow graph*
IR	*Intermediate representation*
SSA	*Static single-assignment*
JIT	*Just-in-time*
PBQP	*Partitioned boolean quadratic problem*
LLVM	*Low-level virtual machine*
PCA	*Principal component analysis*
ANN	*Artificial neural network*
LSTM	*Long short-term memory*
MIR	*Machine intermediate representation*
GGNN	*Gated graph neural network*
GCN	*Graph convolutional network*
ASP	*Answer set programming*
PPO	*Proximal policy optimization*
MCTS	*Monte Carlos search tree*

SUMÁRIO

1 INTRODUÇÃO

Os compiladores constituem uma das classes de programa na Ciência da Computação, cuja principal função é a tradução de código-fonte de uma linguagem de programação para outra. Usualmente, esse processo é empregado para transformar código de alto nível em linguagem de máquina de baixo nível, executável diretamente pelo processador de determinada arquitetura-alvo. [16].

Para atingir esse objetivo, o compilador deve submeter o programa original a uma série de análises — léxica, sintática e semântica — a fim de criar uma representação lógica de sua estrutura e prosseguir com a geração de código. Entre essas duas etapas, é desejável efetuar algumas otimizações a fim de tornar o binário final mais eficiente do ponto de vista de execução e mais coerente com as especificidades da arquitetura-alvo [17].

Dentre as etapas de otimização, destaca-se a alocação de registradores, onde o compilador deve distribuir os registradores para as variáveis utilizadas ao longo do fluxo de execução do programa. Os registradores são componentes microarquiteturais diretamente disponíveis ao processador, e caracterizam-se como as unidades de memória de mais rápido acesso [18]. Contudo, sendo eles um recurso finito, surge a possibilidade de não haverem registradores o suficiente para comportar todos os valores e resultados intermediários em memória de uma única vez [16].

Nesses casos, há a necessidade de se mapear algumas variáveis para a memória principal, e o compilador deve então introduzir no código instruções de acesso à memória para salvar e recuperar os valores lá armazenados. Essas instruções são denominadas código *spill*, ou *spill code* [19]. Um esquema de alocação ideal deve buscar minimizar o tráfego entre a memória principal e a CPU, visto que essas operações, em comparação com acessar os registradores, são significativamente mais lentas e dispendiosas do ponto de vista energético. As operações em memória representam de 50% a 75% dos gastos de energia de um sistema computacional [20], e um alocador sofisticado pode proporcionar uma melhora de até 250% no tempo de execução de um programa comparado a um alocador simples [21].

A abordagem tradicional emprega a coloração de grafo como abstração para representar o processo de atribuição dos registradores físicos para as variáveis virtuais. Nesse modelo, os valores em memória — chamados de registradores virtuais da representação intermediária do código — são expressos como os nós de um grafo a ser colorido; as arestas representam as relações de interferência entre os registradores virtuais, isto é, a presença de intersecções entre o tempo de vida dos valores em memória, enquanto que a coloração em si seria o processo de encontrar registradores físicos disponíveis para cada variável [22].

Contudo, produzir uma alocação eficiente não é uma tarefa trivial. A coloração de grafos é um problema NP-completo, isto é, não há algoritmos determinísticos que a resolvam em tempo polinomial [23], e descobrir a mera quantidade de cores necessárias para colorir um grafo é um problema difícil [24]. Ademais, uma implementação ingênua pode introduzir *spill code* desnecessário ao longo de todo o tempo de vida de um registrador virtual [5], ou realizar escolhas ruins sobre quais variáveis enviar para a memória principal, ao eleger valores frequentemente utilizados no código e que exigirão uma grande quantidade de acessos ou escritas à memória [25].

Ao longo dos anos, soluções para se contornar as dificuldades em se gerar código *spill* de maneira eficiente foram propostas: ajustes no algoritmo original, visando tornar a introdução de instruções `load/store` mais precisa nos pontos de interferência necessários [1, 19, 26, 6], e a utilização de heurísticas para cálculo de custo de *spill*, visando melhorar a qualidade das escolhas de quais variáveis mapear para a memória principal. Infelizmente, as heurísticas dependem de uma boa quantidade de ajustes para propiciar um desempenho adequado. O desenvolvimento de um conjunto de heurísticas para uma arquitetura-alvo específica ainda é feito através de tentativa e erro, e o ajuste fino das funções prioritárias é um processo tedioso [10].

O aprendizado de máquina (*machine learning*), por sua vez, é uma área da inteligência artificial que abrange o desenvolvimento de modelos e agentes inteligentes treinados de maneira algorítmica. Eles realizam tarefas de predição e classificação probabilísticas a partir de parâmetros de entrada, e são calibrados após sucessivas iterações de treinamento, onde os resultados de saída são confrontados com um conjunto de dados previamente conhecidos sobre o problema, de modo a se calcular um erro e ajustar os parâmetros internos do modelo para minimizá-lo [27].

Nas últimas décadas, o aprendizado de máquina e a inteligência artificial como um todo vem avançando consideravelmente, graças ao crescimento do poder computacional das máquinas e conforme cada vez mais dados são gerados e coletados [9]. Esse cenário expande os horizontes para a integração de ambas as áreas, que almeja o desenvolvimento de melhores heurísticas e métodos para otimizar a alocação de registradores empregando técnicas de aprendizado de máquina. Trabalhos vêm sendo publicados explorando a interdisciplinaridade de ambas as áreas e trazendo resultados animadores ou, ao menos, de interesse, como é o caso das pesquisas de Stephenson *et al.*, Das *et al.* e VenkataKeerthy *et al.* [10, 12, 13].

Este trabalho se propõe a investigar a aplicação das técnicas de aprendizado de máquina na alocação de registradores, particularmente sua utilidade na geração de código *spill*. Ele irá contemplar o estado da arte da alocação de registradores e, apoiado no conhecimento acadêmico sobre a área, serão propostos métodos de alocação e heurísticas que utilizam *machine learning*. Utilizando as ferramentas disponíveis na infraestrutura

LLVM — um *framework* utilizado na implementação de compiladores de produção — é esperado desenvolver um modelo de aprendizado que auxilie nas etapas da alocação tradicional, e que tenha um desempenho ao menos comparável ao dos alocadores do *framework* [28].

2 ALOCAÇÃO DE REGISTRADORES

A alocação de registradores é uma etapa de otimização responsável por assinalar registradores físicos às variáveis presentes na representação intermediária (*IR*) produzida pelo *front-end* do compilador. A *IR* é produzida após a conclusão das análises léxica, sintática e semântica no código-fonte, onde o compilador extrai as informações sobre o fluxo de execução e operações aritméticas e consiste, usualmente, de uma simplificação com menos abstrações do programa inicial como, por exemplo, o código de três endereços [16].

O código de três endereços representa o programa como uma sequência de instruções de até três operandos, compostas por expressões de atribuição com operações binárias ou desvios. Para fins de análise, o código é organizado em um grafo onde cada nó contém uma sequência de instruções, chamado grafo de controle de fluxo (*control flow graph*, ou CFG). Ele denota o fluxo de execução do programa e permite ao compilador realizar as análises necessárias para se efetuar otimizações no código, chamadas de *control flow analysis* [29].

A Figura 1 é um exemplo de um trecho de código qualquer, em linguagem C, que apresenta operações aritméticas e uma estrutura de controle. Após a análise dos elementos do código-fonte, o modelo abstrato obtido é similar à árvore da Figura 2. A partir desse modelo, é produzida a representação intermediária equivalente, mostrada na Figura 3.

```
do {
    j = a + a * (c * d - d);
    i--;
} while (i > 0);
i = j;
```

Figura 1 – Exemplo de código C apresentando expressões aritméticas e estruturas de controle.

Devido à natureza da representação intermediária, os valores intermediários computados no cálculo de expressões aritméticas complexas e avaliação de estruturas de controle são expostos na forma de variáveis temporárias, que se juntam às variáveis definidas pelo programador para formar o conjunto dos valores em memória utilizados pelo programa. Esses valores são denominados registradores virtuais, e é trabalho do alocador decidir onde eles serão armazenados [17].

Os registradores virtuais são endereços simbólicos que serão traduzidos em endereços reais após o processo de alocação de registradores. Eles são gerados de maneira incremental pelo compilador de modo a substituir os endereços reais, para possibilitar

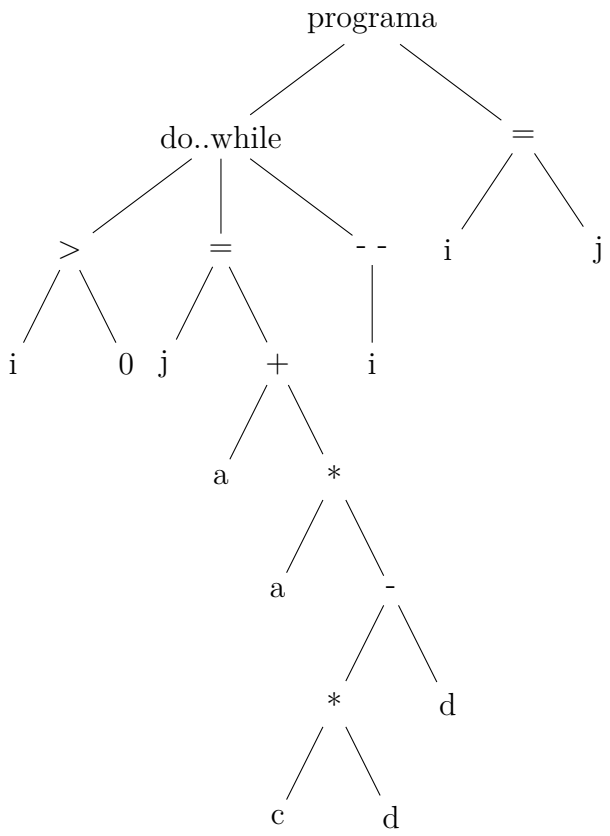


Figura 2 – Árvore sintática abstrata representando o código da Figura 1.

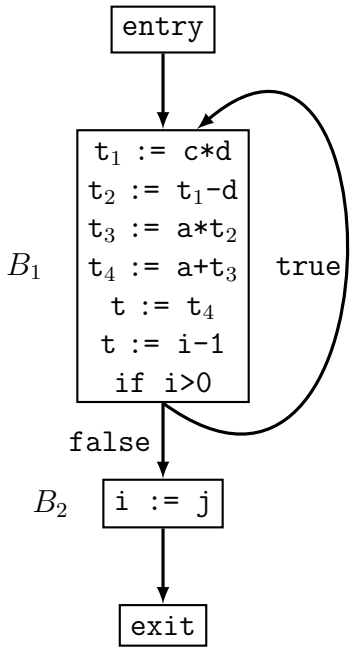


Figura 3 – Grafo de controle de fluxo equivalente gerado a partir da árvore da Figura 2.

uma representação abstrata do programa na forma de código de três endereços e permitir a realização de inúmeras otimizações, sem que a disposição dos endereços de memória seja um entrave para a manipulação da IR [17].

2.1 *Liveness Analysis* e Interferências

Em posse de uma representação intermediária contendo registradores virtuais, o compilador prossegue para a tarefa de mapeá-los para endereços de memória ou registradores físicos. Como dois valores de memória não podem ocupar um único registrador físico ao mesmo tempo, o número de registradores de uso geral disponibilizados pela arquitetura-alvo pode vir a tornar-se um empecilho caso o número de variáveis seja grande demais. Os processadores x86-64 fabricados por companhias como Intel e AMD, por exemplo, possuem comumente 16 registradores de uso geral [30]; arquiteturas RISC como ARMv8, MIPS e SPARC apresentam respectivamente 31, 30 e 32 registradores [31].

Sendo assim, para corretamente alocar os recursos de memória da CPU de modo a comportar as variáveis utilizadas pelo programa, o compilador deve computar os pontos de utilização dos valores ao longo do programa e determinar quando uma variável está viva ou não. Esse processo é denominado análise de longevidade, ou *liveness analysis*, e é uma variação da análise de fluxo de dados também efetuada em outras otimizações.

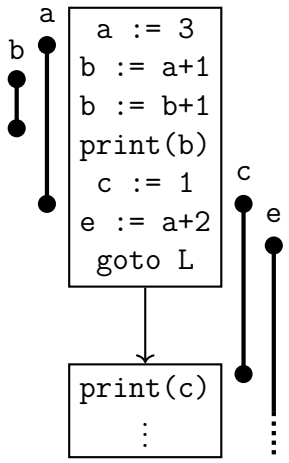


Figura 4 – Dois blocos básicos contendo as variáveis `a`, `b`, `c` e `e`, com seus respectivos *live ranges* indicados.

Uma variável é dita viva no ponto que antecede diretamente a execução de uma instrução se ela contém um valor que será utilizado futuramente ou, em outra palavras, se ela será lida antes da próxima redefinição de seu valor. O conjunto de todos os pontos pelos quais uma variável contendo um valor está viva é chamado de *live range*. Entretanto, uma única variável pode ser quebrada em vários *live ranges*, conforme o valor que ela armazena não será mais utilizado ou é redefinido, dando origem a objetos alocáveis distintos [16].

Quando dois *live ranges* v_i e v_j se sobrepõe em determinado ponto, diz-se que há uma interferência entre eles. Em outras palavras, eles interferem pois a intersecção entre seus conjuntos de pontos não é vazia. Isso significa que ambos os valores presentes em v_i e v_j não podem ser mapeados para o mesmo registrador físico, pois ambos devem ser mantidos em memória integralmente para algum uso posterior. A Figura 4 mostra dois blocos básicos com variáveis e seus respectivos *live ranges*. Nesse exemplo, a variável a interfere com as variáveis b e c.

Se dois *live ranges* v_i e v_j estiverem conectados por uma instrução de cópia na forma $v_i := v_j$ e não interferirem entre si, os dois valores podem ser armazenados no mesmo registrador. Esse ato é denominado coalescimento, ou *coalescing*, e pode promover um ganho significativo na qualidade do código gerado ao eliminar instruções de cópia desnecessárias [32, 1, 19].

2.2 Geração de Código *Spill*

O número de registradores físicos necessários para comportar os *live ranges* em um ponto do código é uma métrica conhecida como pressão de registradores [33]. Quando esse número excede a quantidade de registradores disponíveis para a alocação, faz-se necessário armazenar algum *live range* na memória principal. Para isso, são introduzidas as instruções de acesso a memória `store`, para escrever, e `load` para recuperar um valor em memória.

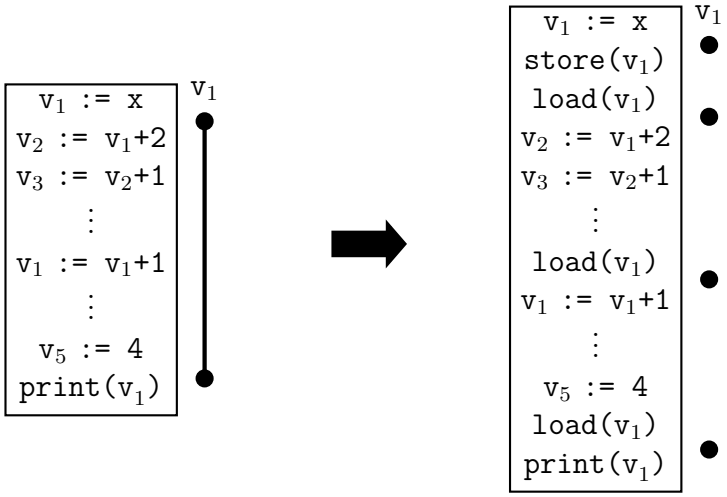


Figura 5 – Exemplo de bloco básico antes e depois do *spill* do *live range* $\mathtt{v}_1$.

Essas instruções são denominadas código *spill*, e tem por efeito fragmentar o *live range* escolhido ao redor do ponto de grande pressão, a fim de reduzi-la [22]. Ao longo do curso de execução de um programa, uma mesma variável pode ser ser armazenada em registradores e posteriormente ser enviada para a memória, e ter seu *live range* particionado [34]. A Figura 5 mostra um bloco básico antes e depois do *spill* da variável v_1, cujo tempo

de vida é decomposto em várias porções menores, aliviando a pressão em determinados pontos.

Esse artifício, no entanto, produz um código mais lento. As instruções de acesso à memória consomem significativamente mais ciclos do que operações aritméticas simples em registradores e desvios. A inserção de *spill code* em locais inapropriados, como em laços de repetição ou trechos que serão frequentemente executados, podem causar um grande *overhead* que derruba a performance do executável gerado. Um alocador eficiente tem a tarefa de minimizar a quantidade de acessos à memória gerados o quanto for possível, e ser preciso em suas escolhas sobre quais variáveis selecionar para *spill* e a colocação das instruções.

2.3 Alocação via Coloração de Grafos

A alocação de registradores por coloração de grafos é a abordagem dominante para o desenvolvimento de alocadores na atualidade. Ela foi implementada pela primeira vez por Chaitin *et al.* em 1980, em um compilador experimental da linguagem PL/I para o IBM System/370 [22], e foi o primeiro método amplamente aplicado para alocação global de registradores, onde os registradores são alocados para toda a unidade de compilação (função) de uma só vez, em vez de alocá-los por bloco [34].

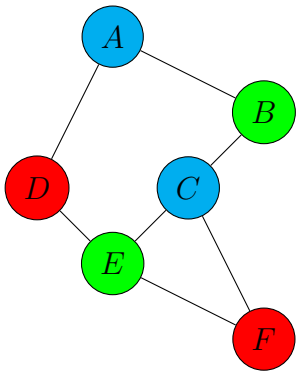


Figura 6 – Exemplo de grafo de interferência colorido com três cores.

Seja $G = (V, E)$ um grafo não-direcionado, onde os vértices $v \in V$ representam os *live ranges* e as arestas $e \in E$, as interferências, tal que $e_n = (v_i, v_j)$ simboliza a existência de uma interferência entre v_i e v_j. O grafo G é dito k-colorível se houver uma função $cor : V \rightarrow \{1, ..., k\}$ tal que $cor(v_i) \neq cor(v_j)$, para todo v_i e v_j em que existir uma aresta $e_n = (v_i, v_j)$. A alocação é reduzida, então, a encontrar o mapeamento dado pela função cor, onde k é o número de registradores físicos presentes na arquitetura-alvo.

Entretanto, computar uma solução ótima não é trivial. A coloração de grafo é um clássico problema NP-completo [23] — determinar a k-colorabilidade de um grafo possui complexidade exponencial e logo torna-se inviável para casos que envolvem muitos registradores virtuais [35, 36], e a mera tarefa de determinar o número mínimo de cores

para se colorir um grafo é um problema NP-*hard* [24]. A Figura 6 exibe uma possível instância de grafo de interferência, formado pelas variáveis A, B, C, D, E e F, e colorido com três cores.

2.3.1 Alocador de *Chaitin*

Chaitin *et al.* [1] formalizaram, em 1982, seu algoritmo de alocação global de registradores via coloração de grafo. A Figura 7 esquematiza a execução do alocador, que consiste em construir um grafo de interferência e manipulá-lo em etapas que estão descritas a seguir:

1. *Renumber* — encontrar todos os *live ranges* dentro do escopo da função ou procedimento, e lhes atribuir um identificador único;

2. *Build* — construir o grafo de interferência $G = (V, E)$, onde cada *live range* se torna um vértice $v \in V$ e as arestas $(v_i, v_j) \in E$ são adicionadas conforme o código é varrido de maneira retrógrada e as interferências são descobertas;

3. *Coalesce* — combinar os *live ranges* que são conectados por uma única instrução de cópia $v_i := v_j$ e não interferem entre si, de modo que os vértices correspondentes no grafo sejam combinados em um único vértice v_{ij}. A instrução de cópia pode ser removida da IR, e as etapas de *build* e *coalesce* devem ser refeitas devido à modificação no código;

4. *Spill cost* — computar o custo de *spill* para cada *live range*. Esse custo é uma estimativa do aumento no tempo de execução se um dado vértice for mapeado para a memória principal, aumento esse que é proporcional ao número de vezes que as instruções de `store` e `load` inseridas serão executadas;

5. *Simplify* — remover os vértices tal que $grau(v) < k$, onde $grau(v)$ é o número de arestas de v e k é número de cores. Isso pode ser feito pois, se um vértice possui um número de arestas menor do que o número de cores, ele certamente é colorível. Sendo assim, após a remoção, o vértice deve ser adicionado a uma pilha auxiliar S, que serve como estrutura de controle da ordem de remoção.

   Se não houverem vértices aptos a serem removidos, algum deve ser escolhido para sofrer *spill*. A prioridade de *spill* é calculada utilizando o custo calculado na etapa anterior, sendo igual a $custo(v)/grau(v)$. O vértice que apresentar o menor custo então é escolhido e a etapa *spill code* é acionada;

6. *Spill code* — o código é alterado com a inserção de instruções de acesso à memória, e o algoritmo deve reiniciar a partir da etapa *renumber*;

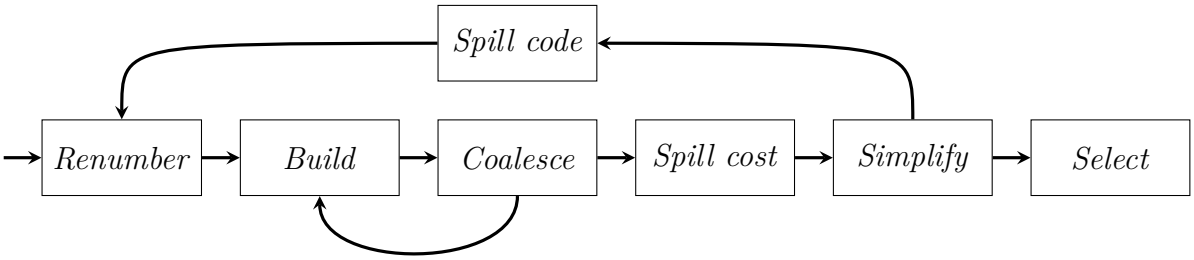


Figura 7 – Esquema do algoritmo de Chaitin [1].

7. *Select* — assinalar cores aos vértices de G, os desempilhando de S um a um, na ordem reversa à que foram removidos na etapa anterior. Se a geração de *spill code* não foi acionada, seguramente todos os vértices receberão uma cor tal que $cor(v_i) \neq cor(v_j)$ onde v_i e v_j são vizinhos.

2.3.2 Alocador de *Chaitin-Briggs*

Briggs [19] propôs uma série de alterações ao algoritmo de Chaitin a fim de corrigir uma série de defeitos que sua versão inicial apresentava. Notavelmente, ele sugeriu um atraso na etapa de geração de *spill code* para produzir alocações mais eficientes e alguns ajustes no processo de coalescimento do alocador, a fim de otimizar o tratamento de instruções de cópia.

Em um trabalho anterior, de 1989 [37], Briggs *et al.* demonstraram que o alocador de Chaitin era ineficiente na sua etapa de *simplify*, ao gerar *spill* em grafos que são trivialmente coloríveis. Um exemplo usado pelo próprio Briggs é o de um grafo em forma de losango, possuindo quatro vértices com duas arestas cada e que é visivelmente colorível para um número de cores $k = 2$, como mostrado na Figura 8. No entanto, o alocador de Chaitin fatalmente enviaria algum registrador virtual para a memória pois, na etapa de *simplify*, não haveriam vértices tal que $grau(v) < 2$ e algum deles e a ativação da etapa *spill code* seria imediata.

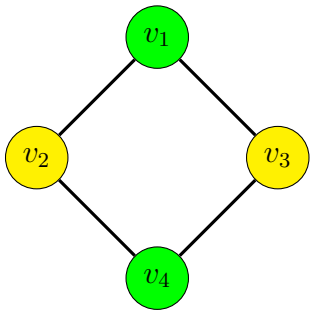


Figura 8 – Grafo 2-colorível que seria incorretamente colorido pelo alocador de Chaitin.

Para solucionar o problema da realização precipitada de *spill*, Briggs propôs adiar a etapa de *spill code* para depois da etapa de *select*. Assim, se por acaso não houverem

vértices aptos a serem removidos durante a fase de simplificação, é escolhido um candidato a *spill* dentre os vértices cujo grau é maior que o número de cores k. Por ser somente um candidato, o alocador ainda deve considerar a possibilidade de colorir esse vértice no futuro e ele pode portanto ser removido do grafo, permitindo destravar o algoritmo e continuar a etapa de *simplify*.

Posteriormente, na etapa de *select*, será avaliada a necessidade de se efetuar *spill* das variáveis candidatas: se ao desempilhar um vértice v tal que $grau(v) > k$ não for possível encontrar uma cor para v, somente então a geração de *spill* code é ativada. Dessa forma, o alocador torna-se capaz de colorir vértices cujo número de arestas é maior do que o número de cores disponíveis, preservando a ordem de coloração produzida pelo alocador de Chaitin. Esssa técnica foi denominada *optimistic coloring*, e é ilustrada pela Figura 9, exibindo o fluxo básico de execução de um alocador ao estilo de Briggs.

Além disso, o algoritmo de Chaitin, na etapa de *coalesce*, poderia transformar um grafo k-colorível em um não-colorível ao combinar dois vértices v_i e v_j em um único vértice v_{ij}, onde $grau(v_{ij}) \geqslant k$. Dessa forma, Briggs *et al.* [26] propuseram o *conservative coalescing*, que consiste em somente coalescer dois vértices se a união de ambos tiver um grau menor do que o número de cores, sendo assim garantidamente colorível.

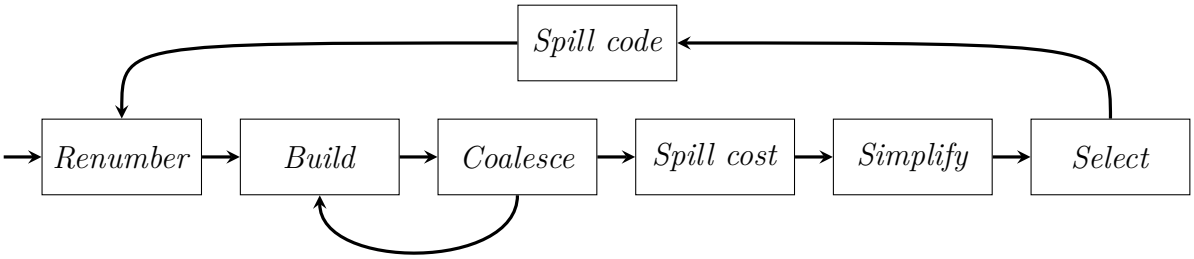


Figura 9 – Esquema do algoritmo de Briggs, com a etapa de geração de *spill code* adiada.

2.3.3 *Iterated Register Coalescing*

Em 1996, George e Appel [32] propuseram uma maneira diferente de coalescer os registradores virtuais: o *iterated register coalescing* é um método mais agressivo de combinar vértices no grafo de interferência, sem adicionar novos *spills* desnecessariamente. Ele consistia em executar as etapas de *simplify* e *coalesce* iterativamente, com a inclusão de uma etapa adicional denominada *freeze*, como descrito a seguir:

1. *Build* — construir o grafo de interferência e categorizar cada vértice como relacionado a `move` ou não. Um vértice relacionado a `move` é aquele que é a origem ou o destino de uma instrução de `move`;

2. *Simplify* — realizar a remoção dos vértices não-relacionados a `move`, com grau menor do que o número de cores;

3. *Coalesce* — combinar vértices ao estilo de Briggs no grafo reduzido obtido da fase anterior. Como os graus de muitos nós já foram reduzidos pela simplificação, provavelmente haverão muito mais vértices válidos para serem coalescidos do que no grafo de inicial. Após dois vértices v_i e v_j terem sido unidos (e a instrução de `move` excluída), se v_{ij} não for mais relacionado a `move`, ele estará apto a ser removido na próxima rodada do *simplify*. Essa etapa e a anterior são repetidas até que restem apenas vértices tal que $grau(v) \geqslant k$ ou relacionados a `move`.

4. *Freeze* — caso nem *simplify*, nem *coalesce* se aplicarem, um vértice de menor grau é escolhido para tornar-se não-relacionado a `move`, e apto a sofrer simplificação. Agora, *simplify* e *coalesce* são retomados.

5. *Select* — assinalar cores aos vértices, de maneira similar ao método de Briggs.

O *iterated register coalescing* representa o estado da arte da alocação de registradores via coloração de grafos, sendo o algoritmo de referência para os compiladores de produção, bem como os ligados à pesquisa [38]. A Figura 10 apresenta um esquema do funcionamento do alocador de George e Appel:

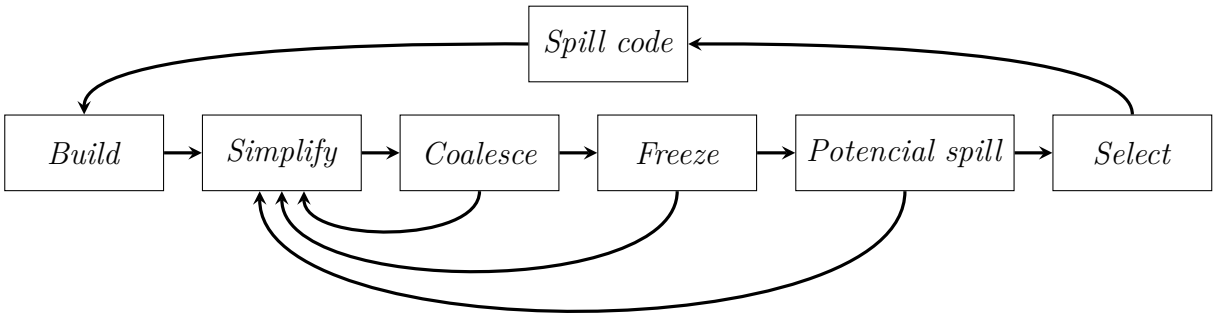


Figura 10 – Esquema do Iterated Register Coalescing, apresentando a etapa de *freeze*.

2.4 Alocação via *Linear Scan*

Um paradigma de alocação de registradores alternativo à coloração de grafos é o da alocação via *linear scan*, que foi proposta pela primeira vez por Poletto e Sarkar [2]. Dados os *live ranges* presentes no escopo de uma função, ao invés de construir um grafo de interferência o algoritmo efetua uma única varredura, na ordem de uma enumeração arbitrária da IR, alocando os registradores físicos através de uma estratégia gulosa.

O algoritmo de *linear scan* é simples, eficiente e produz código comparável ao produzido via coloração de grafos, com a vantagem de propiciar um tempo de compilação consideravelmente mais rápido. Isso pois o algoritmo de *linear scan* apresenta uma complexidade linear, em contrapartida à abordagem por coloração de grafos que tem

complexidade quadrática [39]. Dessa forma, para compiladores onde a velocidade de compilação é uma prioridade, como em compiladores *just-in-time* (JIT) ou interpretadores em tempo real, a alocação via *linear scan* configura-se como a melhor opção.

A técnica de *linear scan* aproxima o conceito de *live range* na forma intervalos vivos, ou *live intervals*. Dada alguma enumeração da representação intermediária, $[i, j]$ é um *live interval* de v se não houverem instruções de número $i' < i$ e $j' > j$, tal que v esteja viva em i' e j'. Podem haver subintervalos de $[i, j]$ onde v não está viva, mas eles são ignorados para todos os fins. Os *live intervals* de um programa podem ser facilmente computados por meio de uma única passagem pelo código, e as interferências entre eles são determinadas pela existência de sobreposições entre os intervalos. O Algoritmo 1 apresenta o pseudocódigo do *linear scan*, como mostrado em Poletto e Sarkar [2].

O algoritmo consiste em percorrer os intervalos $[i, j]$ em ordem crescente a partir do ponto inicial do intervalo i, atribuindo os registradores físicos disponíveis para cada um deles. Uma lista de ativos A é mantida ordenada em ordem crescente dos pontos finais dos intervalos j, e armazena os *live intervals* que se sobrepõe com o ponto atual e foram armazenados em registradores. A cada passo, o procedimento varre A removendo os intervalos "expirados", cujo ponto final precede o ponto inicial do atual intervalo em análise e o registrador correspondente é marcado como livre para ser alocado. O algoritmo então tenta encontrar um registrador disponível para o atual intervalo e, caso não haja nenhum, *spill* deve ser realizado.

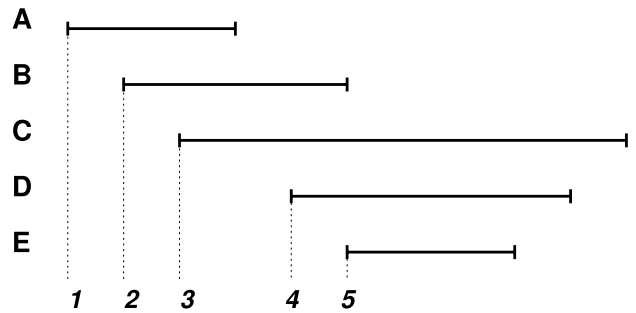


Figura 11 – Exemplo de um conjunto de cinco *live intervals*. Extraído de Poletto e Sarkar [2].

Entretanto, o *linear scan* como proposto por Poletto e Sarkar apresenta duas grandes desvantagens. Em primeiro lugar, devido ao seu aspecto guloso, o algoritmo não leva em consideração "lacunas" nos *live intervals*, isto é, subintervalos onde o valor da variável não é usado e não precisa ser armazenado. Além disso, uma variável enviada para memória permanecerá em memória durante todo o seu tempo de vida [34].

Em 1998, Traub *et al.* [40] propuseram uma versão aprimorada do algoritmo de *linear scan* que realiza a alocação e reescreve o código em uma única varredura no código. No *second-chance binpacking*, o conceito de intervalo é refinado e alcança a mesma precisão dos *live ranges* propriamente ditos, possibilitando aproveitar as lacunas presentes em meio

aos *live intervals*. Ao buscar um registrador para uma nova variável, o alocador pode verificar a existência de lacunas nos intervalos previamente alocados e escolher a menor lacuna de tamanho suficiente para comportar o novo *live interval*, reutilizando o mesmo registrador físico. As Figuras 12 e 13 mostram exemplos do método tradicional e da melhoria de Traub *et al.*, respectivamente.

Algoritmo 1 Alocação de registradores via *linear scan*. Adaptado de Poletto e Sarkar [2].

procedimento LinearScanRegisterAllocation
  $A \leftarrow \{\}$, ordenado em ordem crescente de fim
  para todo intervalo $i \in I$ **faça**
    ExpirarIntervalos(i)
    se $|A| = R$ **então**
      Spill(i)
    senão
      $alocados[i] \leftarrow$ um registrador removido da lista de *livres*
função ExpirarIntervalos(i)
  para todo intervalo $a \in A$ **faça**
    se $fim[a] \geqslant inicio[i]$ **então**
      retorne
    $A \leftarrow A - \{a\}$
    $livres \leftarrow livres \cup \{alocados[a]\}$
função Spill(i)
  $spill \leftarrow$ último intervalo em A
  se $fim[spill] > fim[i]$ **então**
    $alocados[i] \leftarrow alocados[spill]$
    $local[spill] \leftarrow$ lugar na pilha
    $A \leftarrow A - \{spill\}$
    $A \leftarrow A \cup \{i\}$
  senão
    $local[i] \leftarrow$ lugar na pilha

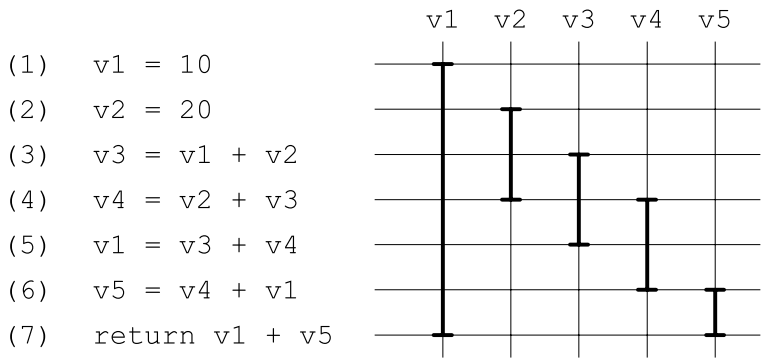


Figura 12 – Exemplo em pseudocódigo mostrando os *live intervals* no *linear scan* tradicional. Figura extraída de [3].

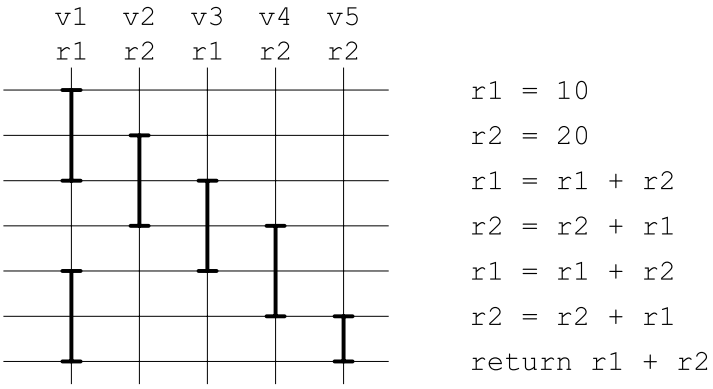


Figura 13 – O mesmo exemplo da Figura 12 com os *live intervals* do *second-chance bin-packing*, e a alocação de registradores final. Extraído de Wimmer [3].

2.5 Alocação via PBQP

Em 2002, Scholz e Eckstein [41] propuseram uma abordagem inovadora para a alocação de registradores valendo-se do PBQP (*Partitioned Boolean Quadratic Optimization Problem*), um problema de otimização que consiste em modelar conjuntos de escolhas como equações booleanas, cada qual com uma lista de custos associados e visando encontrar a sequência de escolhas que produza o resultado com menor custo possível. Esse método de alocação consegue efetivamente combinar as tarefas do alocador e representava as peculiaridades de arquiteturas irregulares em um único problema.

Para cada registrador simbólico, o alocador deve decidir se o armazena em um dos registradores r_1, r_2, ..., r_k, sendo k o número de registradores físicos, ou se o envia para a memória principal. Essa decisão é expressa na forma de uma equação formada por uma soma de variáveis booleanas. Uma das variáveis booleanas representa a decisão de se fazer *spill* do registrador virtual, enquanto as variáveis booleanas restantes representam as decisões de se alocar algum registrador da CPU à variável. A Fórmula 2.1 expressa a equação booleana, onde $x_{spill} \in \{0, 1\}$ representa a decisão de *spill* e $x_i \in \{0, 1\}$ simboliza a alocação de algum dos k registradores.

$$x_{spill} + x_1 + x_2 + \ldots + x_k = 1 \tag{2.1}$$

Como a Equação 2.1 é linear, ela pode ser representada na forma de um vetor booleano $\vec{x_v}$. De maneira análoga, os custos de cada alocação a do conjunto das possibilidades de alocação A, associada aos índices de $\vec{x_v}$, compõe o vetor de custo $\vec{c_v}$. Nesse caso, a atribuição de um registrador físico para o registrador virtual v corresponde a um índice ϕ_a de $\vec{x_v}$ cuja variável booleana correspondente é igual a 1, e f_v é a função do custo de alocação de v dentre o conjunto F_v, que simboliza os custos de se alocar a variável a cada um dos registradores disponíveis. A Fórmula 2.2 apresenta a definição matemática de $\vec{c_v}$:

$$\forall a \in A : \vec{c_v}(\phi_a) = \sum_{f_v \in F_v} f_v(a). \tag{2.2}$$

Sendo assim, tem-se que cada registrador virtual tem a si associado um vetor de custos na forma $\vec{c_v} = [110, 0, 4, \infty, \ldots, \infty]^T$. Nesse exemplo, temos que o primeiro índice é o custo de *spill* da variável; os dois índices seguintes são o custo de alocação para dois registradores físicos hipotéticos. Os índices subsequentes, com custo infinito, representam possibilidades de alocação não permitidas por alguma restrição da arquitetura ou porque já estão ocupadas.

Se dois registradores virtuais u e v forem dependentes entre si devido a alguma interferência nos *live ranges*, estiverem conectadas por uma instrução de cópia ou constituírem um par ditado pela arquitetura, o custo de alocação é expresso pela forma quadrática $\vec{x_u} C_{uv} \vec{x_v}^T$, definido de acordo com a Fórmula 2.3. De maneira análoga, $a_u, a_v \in A$ representam um par de alocações de registradores para u e v respectivamente, e ambos os índices ϕ_{a_u}, ϕ_{a_v} dos vetores de custo individuais de cada variável são levados em conta na obtenção do custo combinado C_{uv}. Por isso, o conjunto dos custos C_{uv} assume a forma da Matriz 2.4, onde os índices de linha e coluna estão associados às decisões individuais em u e v.

$$\forall a_u, a_v \in A : C_{uv}(\phi_{a_u}, \phi_{a_v}) = \sum_{f_{uv} \in F_{uv}} f_{uv}(a_u, a_v) \tag{2.3}$$

$$C_{uv} = \Big[c_{ij}\Big] = \begin{bmatrix} 0 & 0 & \ldots & 0 \\ 0 & \infty & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \infty \end{bmatrix} \tag{2.4}$$

Sendo assim, a tarefa de alocação é expressa na forma de um problema de particionamento quadrático, que consiste em encontrar o esquema de alocação que minimize o resultado da Fórmula 2.5.

$$\min\ f = \sum_{1 \leqslant u < v \leqslant k} \vec{x}_u C_{uv} \vec{x}_v^T + \sum_{1 \leqslant u \leqslant k} \vec{c}_u \vec{x}_u^T. \tag{2.5}$$

Na Fórmula 2.5, os índices u e v representam pares de registradores virtuais e k é o número de registradores virtuais. Devido às propriedades simétricas das formas quadráticas, o resultado da função é uma soma triangular [1]. Observa-se que existe pelo

[1] Soma triangular, ou número triangular, é um número dado pela série $\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n = \frac{(n^2+n)}{2}$ [42].

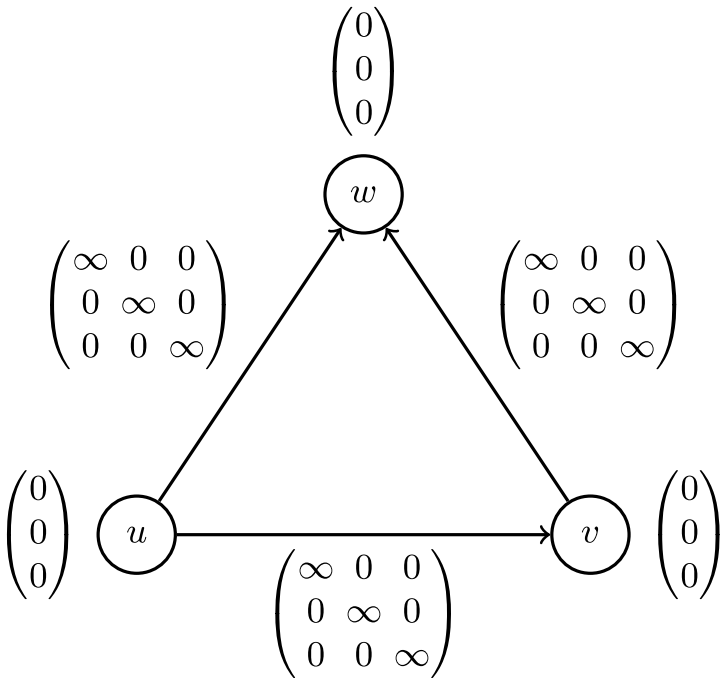


Figura 14 – Grafo de resolução do PBQP considerando 3 registradores virtuais. Extraído de Buchwald *et al* [4]

menos uma solução com custo finito, desde que os custos de *spill* sejam finitos. Isto se deve ao fato de que todos as variáveis podem sofrer *spill*, embora não seja uma boa solução.

Ainda que o problema possa ser expresso formalmente, uma modelagem baseada em grafos é mais intuitiva. Nesse contexto, cada vértice do grafo possui um vetor de escolha associado, que atribui a cada alternativa seu custo de execução, e para cada nó apenas uma alternativa pode ser selecionada. As interdependências são modeladas por arestas direcionadas com uma matriz associada, que contém os custos das combinações de alternativas. A Figura 14 mostra um exemplo de representação de um problema PBQP via grafo. Cada vértice u, v e w possui três alternativas que representam suas cores possíveis. Devido às matrizes de arestas, cada solução possível deve selecionar cores diferentes para nós adjacentes e, portanto, representa uma 3-coloração válida [4].

O PBQP é um problema NP-completo, de maneira similar à coloração de grafos. No entanto, soluções quase-ótimas podem ser obtidas através de técnicas como programação dinâmica e a aplicação de heurísticas de simplificação, que permitem reduzir uma instância do problema de tal modo que a melhor resolução torne-se trivial. Ao retropropagar as reduções, a seleção da instância menor pode ser estendida para uma seleção da instância original do PBQP. Originalmente, foram propostas quatro reduções [43, 44]:

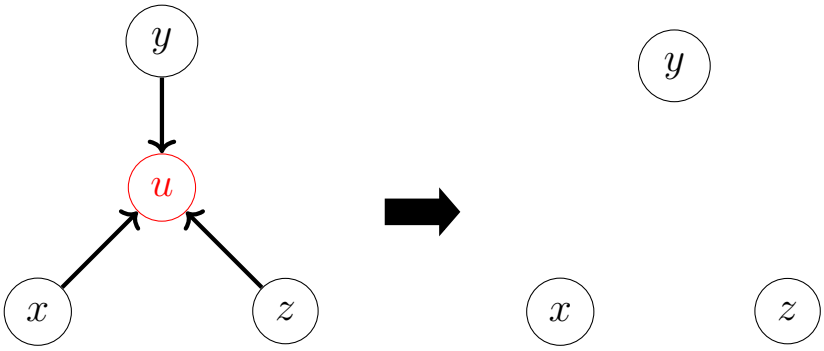
1. *RE* — remoção de arestas independentes, possuindo uma matriz de custos que pode ser decomposta em dois vetores $\vec{u}$ e $\vec{v}$, ou seja, cada entrada da matriz c_{ij} tem custos $u_i + v_j$. Essas arestas podem ser removidas ao se somar $\vec{u}$ e $\vec{v}$ aos vetores de custo dos vértices de origem e de destino, respectivamente. Se isto produzir custos

vetoriais infinitos, a alternativa correspondente (incluindo linhas/colunas da matriz) é eliminada;

2. *RI* — remoção de vértices de grau um, somando os custos no vértice adjacente;

3. *RII* — remoção de vértices de grau dois, após os custos serem contabilizados na matriz de custos da nova aresta entre os dois vizinhos. Se necessário, a aresta é criada primeiro;

4. *RN* — remoção de vértices de grau três ou maior. Dado um vértice u, é escolhido um mínimo local e os custos de u são distribuídos entre seus vizinhos. Em seguida, u é removido e as arestas são removidas usando *RE*.



As reduções *RE*, *RI*, e *RII* produzem subdivisões da instância original de igual custo mínimo, permitindo estender a seleção dos problemas menores para o problema original sem comprometer a qualidade da solução. Se o grafo inteiro puder ser reduzido dessa maneira, então é possível encontrar uma solução ótima para dada instância do problema. Se *RN* for aplicada, no entanto, as subpartes produzidas perdem a precisão de representação do problema e a solução final obtida torna-se quase-ótima, onde a proximidade de uma hipotética solução ótima é garantida através da heurística de escolha do mínimo local como alternativa.

Hames e Scholz [45] propuseram, em 2006, uma heurística alternativa para a redução *RN* que elimina a escolha de uma alternativa na etapa de redução, efetivamente a postergando para a fase de retropropagação das soluções. Além disso, eles introduziram

um abordagem *branch-and-bound* que consiste em utilizar de um limiar para dividir as possíveis combinações de escolhas, de tal modo que as combinações que certamente não são ótimas são descartadas e nunca são avaliadas pelo alocador. Essas heurísticas garantem resolução em tempo linear, e as reduções tornam a alocação via PBQP especialmente poderosa em instâncias que produzem grafos de interferência esparsos, pois permitem a alocação de maneira trivial.

3 MINIMIZAÇÃO DE *SPILL CODE*

Mesmo com a utilização de algoritmos refinados de alocação, em vários casos a introdução de *spill code* no código resultante é factualmente inevitável. Sendo assim, um bom alocador deve produzir uma alocação visando minimizar o tráfego de dados entre a memória principal e o processador, devido ao grande atraso provocado pela execução de instruções de acesso à memória. Essas operações são responsáveis pela maior parte do gasto energético em sistemas computacionais [20], e a alocação de registradores pode impactar o tempo de execução de um programa em até 250% [21].

Dentre os problemas enfrentados pelos desenvolvedores de compiladores, há a notável tarefa de se decidir qual registrador virtual será enviado para a memória. A escolha deve ser tomada de modo a reduzir a pressão de registradores nos pontos problemáticos do código, preferencialmente tornando-o inteiramente alocável e resolvendo o problema com o *spill* de uma única variável. No entanto, a escolha de uma variável frequentemente acessada, que está dentro de um laço de repetição ou é usada como controle, pode acidentalmente produzir muito mais acessos à memória no programa [1, 25].

Além disso, há espaço para diferentes análises na tarefa de se estimar custos de *spill*. Diferentes métodos de se definir o custo das operações de acesso à memória e calcular o custo associado à escolha de cada variável são possíveis, e podem produzir resultados diferentes considerando a especificidade arquitetura-alvo. Sendo assim, os desenvolvedores de compiladores dependem de experimentação e "tentativa e erro" para ajustar suas heurísticas de alocação [10].

3.1 Heurísticas de Chaitin

A proposta inicial do primeiro alocador via coloração de grafos, feita por Chaitin *et al.* em 1981, não continha heurísticas significativas para otimizar a introdução de *spill code*. Inicialmente, a prioridade de escolha para a realização de *spill* era dada às variáveis denominadas *pass-through* — isto é, que estão vivas na entrada dos blocos básicos e não são usadas nem redefinidas posteriormente. Instruções de `store` eram inseridas após toda redefinição do *live range* escolhido, e os `loads` eram necessários antes de qualquer uso [22].

Sendo assim, em um trabalho posterior de 1982 [1], Chaitin formalizou heurísticas para a tomada de decisões de *spill* que almejavam reduzir a quantidade de acessos a memória em tempo de execução. O custo de *spill* de um registrador virtual é definido como o aumento no tempo de execução caso ele resida em memória, que corresponde ao número de pontos de definição mais o número de usos do valor. Por suas vezes, as definições

e usos são ponderados levando em conta suas frequências estimadas de execução.

O custo de *spill* associado a um *live range* v no alocador de Chaitin é dado pela Fórmula 3.1, onde i é uma instrução contida no conjunto de instruções da IR do programa I, que manipula v através de um uso ou uma redefinição, representados respectivamente pelos predicados $use_v(i)$ e $def_v(i)$. A função $profundidade(i)$ expressa o nível de aninhamento de i dentro de laços de repetição.

$$custo(v) = \sum_{\substack{i \in I \\ use_v(i) \vee def_v(i)}} 10^{profundidade(i)} \tag{3.1}$$

Os custos são computados quando, durante a etapa de *simplify*, não são encontrados mais vértices aptos a serem removidos do grafo de interferência e o algoritmo trava. Em seguida, o alocador define as prioridades para a escolha de qual variável sofrerá *spill* utilizando a Fórmula 3.2. Para cada *live range*, o custo é dividido pelo grau do vértice correspondente no grafo e o vértice com menor valor ponderado é escolhido para sofrer *spill* em todo o código [25].

$$h(v) = \frac{custo(v)}{grau(v)} \tag{3.2}$$

Chaitin [1] ainda introduziu o conceito de proximidade entre instruções: duas instruções são ditas próximas se nenhum *live range* for eliminado entre elas e, consequentemente, nenhum registrador for liberado para alocação nesse intervalo. Esse princípio fundamenta um conjunto de três diretrizes que buscam evitar a inserção de *reloads* desnecessários no código. São elas:

1. Se uma definição e um uso são próximos, não é necessário adicionar um `load` antes do uso. Isso ocorre porque, 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;

2. Se duas instruções de uso são próximas, é necessário introduzir um `load` somente antes do primeiro uso. De maneira análoga à diretriz anterior, o mesmo registrador pode ser reaproveitado na segunda instrução;

3. Se a primeira definição e o último uso de um *live range* são próximos, então o custo de *spill* do registrador virtual é considerado infinito. Como nenhum registrador é liberado, enviar a variável para memória não tornará o programa colorível.

3.2 Heurísticas de Bernstein

Em um trabalho de 1989, Bernstein *et al.* [25] pontuaram uma série de problemas com o alocador de Chaitin. Em primeiro lugar, a heurística empregada na decisão de *spill* pode não ser a mais eficiente em todas as instâncias de alocação. Além disso, o alocador de Chaitin insere instruções `store/load` desnecessariamente em todos os pontos do antigo *live range*, sendo que elas podem somente ser necessárias em pontos de grande pressão de registradores.

Sendo assim, o trabalho trouxe algumas técnicas e novas heurísticas visando expandir as contribuições de Chaitin. Essas melhorias se mostraram capazes de reduzir a quantidade de código *spill*, na média, em 6% e 12%, chegando até 30% em alguns casos.

3.2.1 *Best-of-three*

Bernstein *et al.* notaram que, a respeito da heurística de decisão de *spill*, não se pode fazer uma afirmação absoluta sobre o desempenho de uma função heurística simples em relação a outra para todos os programas. Ao contrário, diferentes instâncias de problemas de alocação podem exigir estratégias distintas para a obtenção de melhores resultados. Eles propuseram comparar o desempenho médio de diferentes fórmulas, mantendo em mãos um seleto número funções heurísticas que vislumbrem as diferentes causas da pressão de registradores, e escolhendo a melhor delas para tomar as decisões de *spill*.

A heurística h_1 (Fórmula 3.3) tem em vista a estratégia original de Chaitin, que consiste em escolher um vértice com baixo custo e alto grau. Realizar *spill* de uma variável com alto grau reduz o grau de muitos outros vértices no grafo de interferência, tornando mais provável que outros vértices se tornem desobstruídos.

$$h_1(v) = \frac{custo(v)}{grau(v)^2} \tag{3.3}$$

Alternativamente, a abordagem escolhida pode ser a de se fazer *spill* do registrador virtual que exerça a maior pressão de registradores sobre o código. Sendo assim, é introduzido o conceito de área de uma variável, computado através da Fórmula 3.4:

$$\acute{a}rea(v) = \sum_{\substack{i \in I \\ v \text{ vivo em } i}} 5^{profundidade(i)} largura(i) \tag{3.4}$$

onde $largura(i)$ é o número de variáveis vivas no momento de execução da instrução i. De maneira intuitiva, $\acute{a}rea(v)$ expressa o impacto de v para a pressão de registradores global. A escolha de uma variável com alta área causa uma redução considerável na pressão em todo o programa, e facilita a coloração. As heurísticas seguintes h_2 e h_3 (Fórmulas 3.5 e 3.6) se fundamentam nesse princípio.

$$h_2(v) = \frac{custo(v)}{área(v)grau(v)} \tag{3.5}$$

$$h_3(v) = \frac{custo(v)}{área(v)grau(v)^2} \tag{3.6}$$

O algoritmo proposto por Bernstein *et al.* efetua efetua a coloração do grafo de interferência múltiplas vezes, para cada heurística h_i disponível. O resultado final escolhido é o da heurística que produza o menor custo total de *spill*. Essa técnica, apelidada de "*best-of-three*", superou as abordagens anteriores consideravelmente, sob penalidade de um tempo de compilação ligeiramente maior.

3.2.2 *Coloração gulosa*

Bernstein *et al.* [25] também introduziram heurísticas de coloração que se baseiam em adotar uma estratégia local para cada região do programa. É sabido que o grafo de interferência pode conter diversos subgrafos com diferentes características, sendo que alguns deles podem ser coloridos em tempo polinomial [46, 47]. Dessa maneira, diferentes formas de coloração podem ser adotadas em diferentes porções do código, e os resultados combinados para compor a alocação final.

O algoritmo de coloração de por Bernstein *et al.* realiza a etapa de *simplify* removendo sempre o vértice apto de maior grau. Logo, na etapa de *select*, os registradores são alocados em ordem crescente de grau para cada *live range*. Ademais, após a k-colorabilidade do grafo de interferência ter sido assegurada, um critério secundário pode usado para colori-lo com, geralmente, menos cores do que em uma atribuição aleatória. Em certas situações, isso pode melhorar o código resultante, como:

- A alocação produziu *spills* a mais, e torna-se possível colorir o grafo com menos do que k cores. Nesse caso, as $k - n$ decisões de *spill* mais custosas da iteração anterior podem ser desfeitas;
- Pequenos procedimentos chamados frequentemente, onde uma utilização de registradores econômica pode reduzir o *overhead* das chamadas;
- Expansões *inline* de funções, onde a decisão de expandir ou não é influenciada pela quantidade de registradores utilizados na sub-rotina.

3.2.3 *Cleaning*

Bernstein *et al.* [25] resolveram o problema da inserção desnecessária de instruções `store`/`load` através da técnica denominada pelos autores como "*cleaning*", ou limpeza. Esse método consiste em, quando uma variável for enviada para a memória, inserir somente

uma instrução `store` e um `load` por bloco básico. Ao examinar um bloco em busca de valores que sofreram *spill*, os acessos à memória são introduzidos apenas no primeiro uso ou definição do *live range*. Em seguida, a variável é renomeada em cada bloco básico em que é usada, fragmentando o tempo de vida original.

Os autores observaram que o *cleaning* é bem sucedido em reduzir o número total de instruções de acesso à memória, especialmente nas primeiras duas iterações de coloração/*spill*, e quando nenhuma restrição é imposta à profundidade dos blocos básicos nos quais a limpeza é realizada ou aos registradores para os quais a limpeza é aplicada. O pseudocódigo mostrando o fluxo de execução da técnica de Bernstein *et al.* com a realização do *cleaning* encontra-se no Algoritmo 2.

Algoritmo 2 Algoritmo de coloração utilizando *best-of-three*. Extraído de Bernstein *et al.* [25]

Dados: R: conjunto de registradores físicos, G: grafo de interferência, S_i: lista de *spill* associada à heurística h_i, h_i: heurística.
procedimento ColorAndSpill
  para todo método heurístico h_i **faça**
    enquanto G não é vazio **faça**
      se existe um v no grafo G tal que $grau(v) < |R|$ **então**
        escolha o v com maior grau (tal que $grau(v) < |R|$)
        $G \leftarrow G - \{v\}$
      senão
        escolha um v tal que min $h_i(v)$
        $S_i \leftarrow S_i \cup \{v\}$
        $G \leftarrow G - \{v\}$
    restaure G
  escolha a heurística h_i com o menor $custo(S_i)$
  se nenhum vértice tiver sofrido *spill* **então**
    pinte os vértices em ordem reversa da remoção de G
  senão
    para todo vértice $v \in S_i$ **faça** Spill(v)
    realize *"cleaning"* nos blocos básicos
    reconstrua o grafo G

3.3 *Spilling* por Região de Interferência

Em 1997, Bergner *et al.* publicaram um artigo [5] pontuando que as heurísticas prévias ainda produziam *spill* code desnecessariamente, pois as instruções de acesso à memória eram inseridas ao longo de todo o tempo de vida que era escolhido para *spill*. Em resposta a esse problema, Bergner *et al.* introduziu o conceito de *spilling* por região de interferência, que consistia em limitar as alterações no código somente às regiões de sobreposição entre os *live ranges*, nos pontos de alta pressão de registradores.

Nessa abordagem, uma região de interferência entre dois registradores virtuais é dita como a porção do programa onde ambos estão vivos simultaneamente, e elas são diretamente representadas no grafo de interferência na forma das arestas. Em adição às técnicas de minimização de *spill code* propostas por Chaitin [22], Bernstein [25] e Briggs [26], o alocador de Bergner limita a introdução de *reloads* somente aos usos da variável dentro da região de interferência. No caso em que o *live range* é utilizado novamente após a região de interferência, deve-se introduzir um *reload* adicional para recarregar o valor de volta para um registrador físico.

```
A = input();
B = A + 1;
if (A) {
     C = A + 2;
     B = A + C;
     if (C) {
          B = B + C;
          C = B + C;
     }
     A = B + C;
}
D = A + B;
```

Figura 15 – Exemplo de programa em pseudocódigo extraído de Bergner *et al.* [5].

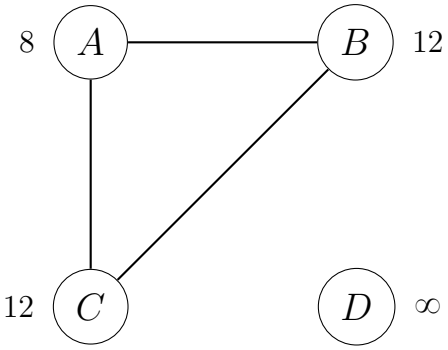


Figura 16 – Grafo de interferência correspondente ao código da Figura 15, com os custos de *spill* indicados.

Para escolher quais regiões de interferência sofrerão *spill*, a estratégia desenvolvida por Bergner é aplicada durante a etapa de *select* de um alocador ao estilo de Briggs, durante a reintrodução dos vértices no grafo e concomitante alocação de cores. Ao se tentar reintroduzir um vértice v candidato a *spill* — o que não será inteiramente possível devido ao número de interferências de v, suas arestas são agrupadas em conjuntos, cada conjunto contendo as arestas que se conectam a vértices de uma mesma cor. O alocador deve então tentar colorir v, não inserindo no grafo as arestas do conjunto cuja cor já está atribuída a algum dos seus vizinhos.

O custo de se fazer *spill* de v em cada região de interferência é calculado levando em conta a quantidade estimada de instruções `store`/`load` que serão introduzidas no código, e a cor selecionada para o vértice candidato deve ser a que minimize o total desse custo. Uma vez que a cor é definida, as regiões de interferência cujas arestas correspondentes não puderam ser inseridas no grafo são escolhidas para sofrerem *spill*. A Figura 17 demonstra esse processo ao tentar colorir o grafo da Figura 16, onde o vértice candidato A recebe a mesma cor de C, impedindo que a aresta AC seja introduzida no grafo. Ela é então descartada, e A sofrerá *spill* na região de interferência entre A e C.

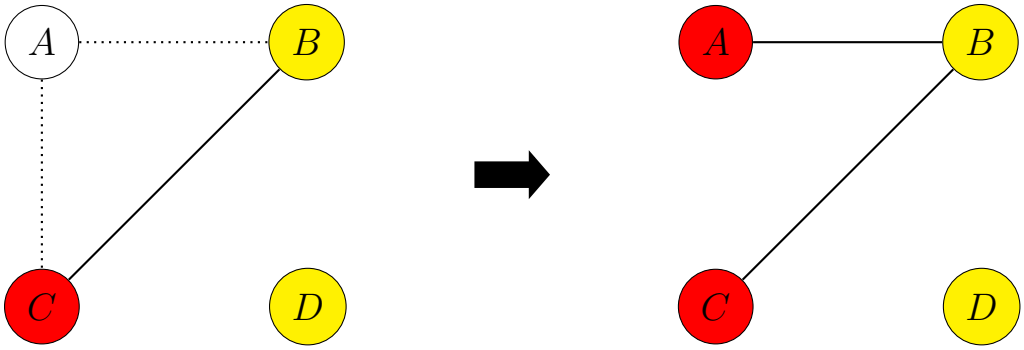


Figura 17 – Coloração do exemplo da Figura 16, considerando a estratégia de *spilling* por região de interferência. Adaptado de Bergner *et al.* [5].

A comparação entre o código gerado pela técnica tradicional e o pelo *spilling* por região de interferência é exibida na Figura 18. A estratégia de Bergner *et al.* apresentou altas taxas de sucesso em experimentos, conquistando uma redução média de 33,6% no número de instruções de acesso a memória executadas dinamicamente, atingindo a faixa de 75% em alguns casos [5]. Essa técnica superou de longe as heurísticas anteriores, e representou um avanço significativo na minimização de *spill code*.

```
A = input();                    A = input();
store A;                        store A;
B = A + 1;                      B = A + 1;
if (A) {                        if (A) {
    load A1;                        load A1;
    C = A1 + 2;                     C = A1 + 2;
    B = A1 + C;                     B = A1 + C;
    if (C) {                        if (C) {
        B = B + C;                      B = B + C;
        C = B + C;                      C = B + C;
    }                               }
    A2 = B + C;                     A = B + C;
    store A2;
}                               }
load A3;
D = A + B;                      D = A + B;
```

Figura 18 – Comparação entre os resultados da técnica tradicional e do *spilling* por região de interferência. Na esquerda, `A` sofre *spill* em todo o código, enquanto na direita a inserção de *spill code* ocorre somente na interferência entre `A` e `C`. Adaptado de Bergner *et al.* [5].

3.4 Rematerialização

Em seu trabalho de 1982, Chaitin *et al.* [1] já haviam demonstrado que é preferível inserir instruções para recalcular certos valores ao invés de armazená-los em memória, devido ao custo mais baixo de fazê-lo. Essa técnica foi denominada "*rematerialization*", ou rematerialização, e representa uma alternativa à geração de *spill code* em situações onde a pressão de registradores ultrapassa o número de registradores físicos disponíveis. Posteriormente, Briggs *et al.* [26] refinaram essa técnica, introduzindo novos conceitos e propondo uma metodologia que permite uma otimização mais profunda.

Chaitin *et al.* mostraram que certos valores podem ser recalculados com uma única instrução adicional, caso os operandos necessários estejam sempre disponíveis. Eles denominaram esses casos excepcionais como valores *"never-killed"*, e argumentaram que é mais barato recomputar tais valores do que armazená-los e constantemente acessar a memória para recuperá-los. Na prática, boas oportunidades para a rematerialização incluem `loads` imediatos de constantes e o cálculo de endereços de memória, com ou sem *offset*. Entretanto, o alocador de Chaitin só pode rematerializar *live ranges* que assumem um único valor ao longo de toda sua extensão, não sendo capaz de lidar com casos mais complexos.

Sendo assim, em 1992 Briggs *et al.* [26] expandiram a técnica de Chaitin para tratar *live ranges* multivalorados. Sua abordagem consistia em dividir os tempos de vida para cada valor que assumem ao longo da execução de um programa, efetivamente convertendo

a representação intermediária para a forma SSA (*static single-assignment*). Na forma SSA, cada variável é definida uma única vez antes de ser obrigatoriamente usada, e por isso possui apenas um valor ao longo do seu tempo de vida.

No processo de conversão para a forma SSA, os *live ranges* dão origem vários tempos de vida menores definidos uma única vez, a partir de um valor simples ou uma função-ϕ — artifício que representa a utilização de um dentre múltiplos valores possíveis em tempo de execução. A Figura 19 apresenta um exemplo de código nas formas tradicional e SSA, onde a variável `w` é definida a partir de `y`, que recebe valores diferentes em cada caso de uma ramificação condicional. Sendo assim, deve ser criada uma nova variável y_3 que pode receber tanto o valor de y_1 quanto y_2, a depender de qual bloco condicional será acessado em tempo de execução.

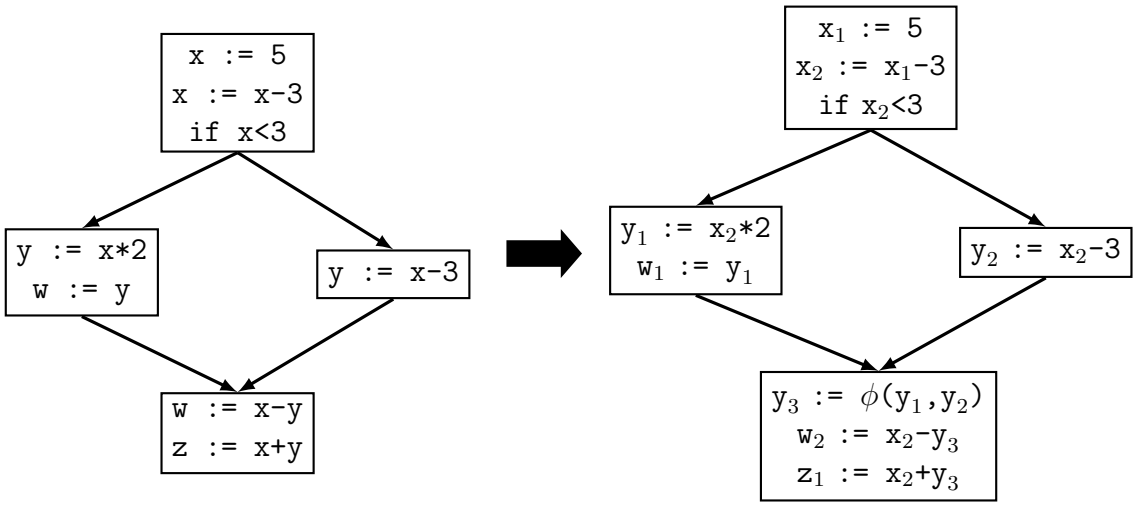


Figura 19 – Exemplo de código na forma tradicional e após a conversão para forma SSA.

Em seguida, as instruções de definição dos valores são rotuladas como candidatas ou não à rematerialização. Para isso, Briggs *et al.* empregaram um algoritmo semelhante ao *constant propagation* de Wegman e Zadeck [48], para propagar as rotulações pelo grafo de controle de fluxo. Os rótulos consistem de:

- $\top$ — significa que nenhuma informação é conhecida. Um valor definido por uma instrução de cópia ou uma função-ϕ recebem este rótulo de imediato;
- *inst* — valor definido por uma instrução adequada (*never-killed*. Na implementação é um ponteiro para a própria instrução;
- $\bot$ — valor que deve sofrer *spill*. Valores inapropriados para a rematerialização recebem este rótulo de imediato.

Além disso, quando o rótulo de uma variável é propagado para as instruções que a utilizam ocorre uma operação de *"meeting"* ou encontro, denotada pelo símbolo $\sqcap$. Ela é

efetuada entre rótulos, e é definida matematicamente através das Fórmulas 3.7, 3.8 e 3.9:

$$x \sqcap \top = x \tag{3.7}$$

$$x \sqcap \bot = \bot \tag{3.8}$$

$$inst_i \sqcap inst_j = \begin{cases} inst_i, & \text{se } inst_i = inst_j \\ \bot, & \text{se } inst_i \neq inst_j \end{cases} \tag{3.9}$$

onde x é qualquer rótulo, e a comparação $inst_i = inst_j$ é realizada operando a operando. Os valores são todos inicializados como $\top$, e durante a propagação valores definidos por uma instrução de cópia terão seus rótulos reduzidos para *inst* ou $\bot$. Valores definidos por funções-ϕ serão reduzidos para *inst* se e somente se todos os valores o atingindo tiverem rótulos equivalentes; caso contrário, eles recebem $\bot$.

Concluída a propagação, os nós-ϕ devem ser removidos e os valores renomeados para que o programa executável seja produzido. Valores rotulados como *never-killed* são rematerializados no código final e possíveis divisões desnecessárias dos *live ranges*, remanescentes da forma SSA, são removidas através de processos presentes em um alocador estilo Briggs [19], como coalescimento e coloração com a mesma cor de variáveis unidas por cópia.

O algoritmo de Briggs *et al.* foi testado em 70 benchmarks, apresentando uma redução considerável de *spill code* em 28 dos casos, com melhorias que ultrapassaram os 20%. Os autores também observaram uma redução nas instruções de `store`, `load` e de cópia, o que indica que as heurísticas de remoção de divisões desnecessárias são adequadas [26].

3.5 Coloração de grafos hierárquica

Em 1991, Callahan e Koblenz [49] descreveram uma técnica de alocação de registradores que emprega heurística hierárquica de coloração de grafos, onde o programa é particionado em uma série de porções que constituem uma estrutura de árvore e que refletem a estrutura de controle do código. Então, para cada porção é realizada a coloração de grafos de maneira local, resultando numa alocação sensível aos padrões de utilização das variáveis em cada região do programa.

O princípio da abordagem de Callahan e Koblenz consiste em representar o fluxo de repetição e ramificação nos blocos básicos na forma de *"tiles"* que na prática constituem subárvores. Sendo o grafo de controle de fluxo (CFG) representado pela tupla $G = (B, E, start, stop)$, onde B é o conjunto dos blocos básicos, E é o conjunto de arestas entre os blocos e $start, stop \in B$ representam respectivamente os pontos de início e fim do fluxo de execução, é definido uma árvore hierárquica T.

Cada nó $t \in T$ é um membro da família de subconjuntos de B tal que $blocos(t) = \{b_0, b_1, \ldots, b_n\}$ e $arestas(t) = \{e_0, e_1, \ldots, e_n\}$, onde b são os blocos básicos pertencentes a t, mas que não pertencem a nenhuma subárvore de t, e e são as arestas dos blocos em $blocos(t)$. Cada par de elementos em $t_1, t_2 \in T$ é ou disjunto, ou t_1 é subconjunto próprio de t_2 e não há nenhum outro t tal que $t_1 \subset t \subset t_2$; nesse caso, dizemos que t_1 é nó-filho de t_2 e t_2 é nó-pai de t_1. Há também em todo programa uma raiz t_0 tal que $blocos(t_0) = \{start, stop\}$. A adição de novos nós na árvore do programa ocorre nas ramificações criadas por estruturas condicionais e em laços de repetição, como exemplificado na Figura 20.

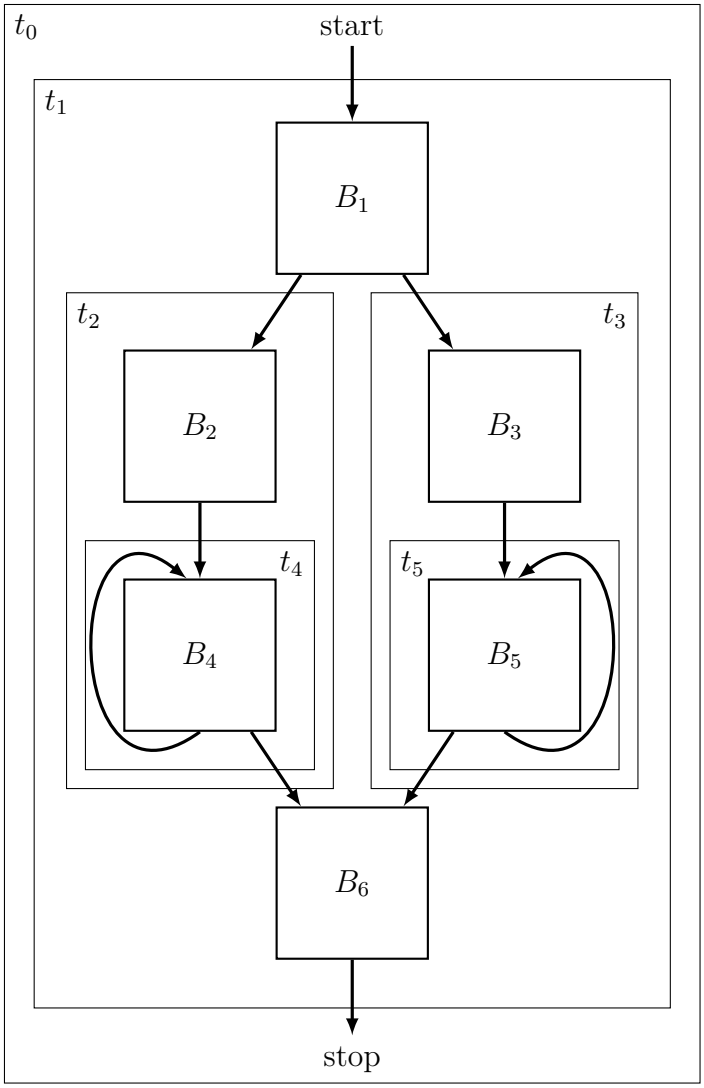


Figura 20 – Exemplo de CFG particionado em porções correspondentes aos nós $t_0, t_1, \ldots, t_5$ da árvore hierárquica.

Uma vez que a representação intermediária é organizada na forma da árvore hierárquica, a coloração de grafos é efetuada para cada nó de maneira recursiva em pós-ordem, ou *"bottom-up"*. Para cada nó t, o grafo é montado contendo somente as variáveis cujo *live range* é definido e usado inteiramente em $blocos(t)$, que são denominadas variáveis locais, e a coloração é feita de maneira convencional seguindo um algoritmo ao estilo de

Briggs [37]. Para todos os fins, as variáveis globais a t — isto é, que são definidas em níveis superiores da árvore e estão vivas na entrada dos blocos básicos de um nó — são ignoradas, pois serão tratadas na alocação de algum nó ancestral de t. Uma vez concluída a coloração, as variáveis recebem registradores físicos diretamente ou pseudoregistradores, que funcionam como alcunhas para grupos de registradores da arquitetura-alvo.

Ao retornar para a chamada anterior da recursão, o algoritmo prossegue para colorir o grafo do nó-pai t_{pai} incluindo um conjunto de novas variáveis, cada uma delas sendo o coalescimento de todas as variáveis de t que foram alocadas a um registrador distinto. Elas são denominadas "variáveis de resumo", e servem o propósito de indicar quais registradores já foram aproveitados na subárvore com raiz em t. Sendo assim, uma aresta é inserida no grafo de interferência de um nó se:

- Duas variáveis locais ao nó interferem em algum ponto de seus blocos;
- Uma variável global interfere com uma variável de resumo ou outra variável global;
- Uma variável está viva em uma subárvore de t mas não consta nas variáveis de resumo vindas da subárvore;
- Uma variável de resumo conflita com outras variáveis de resumo da mesma subárvore.

Uma vez que a raiz da árvore é colorida, a atribuição final de registradores físicos ocorre de maneira *"top-down"*, assim como a inserção de instruções de *spill*. O *spill code* é, de maneira geral, inserido nas entradas e saídas dos blocos básicos e consiste de instruções de acesso a memória para variáveis locais que sofreram *spill* em um nó, instruções de cópia para variáveis que receberam registradores diferentes no nó pai e no nó filho, além de instruções `load` para recarregar valores que receberam um registrador no nó pai mas sofreram *spill* no nó filho.

Para decidir quais *live ranges* devem receber um registrador é empregada uma heurística de decisão de *spill* similar à de Chaitin [1], com a adição de uma nova fórmula para o cálculo de custo desenvolvida por Callahan e Koblenz. Supondo custo unitário para as operações de acesso à memória, são definidas as métricas $peso\,local_t$ (Fórmula 3.10), correspondente ao custo de se manter um valor em registradores físicos levando em conta $blocos(t)$, e $transferência_t$ (Fórmula 3.11), que é o custo de se inserir *spill code* nas entradas e saídas dos blocos de t.

$$peso\,local_t(v) = \sum_{b \in blocos(t)} prob(b) ref_b(v) \tag{3.10}$$

$$transferência_t(v) = \sum_{e \in arestas(t)} prob(e) viva_e(v) \tag{3.11}$$

As funções $prob(b)$ e $prob(e)$ expressam respectivamente a probabilidade de um bloco e de uma aresta serem executados; $ref_b(v)$ denota o número de referências a v no bloco b e $viva_e(v)$ é uma função booleana que indica se v está viva na aresta e. Sendo assim, a heurística de custo que fundamenta a decisão de qual variável deve sofrer *spill* é dada pela função $peso_t$, computada segundo a Fórmula 3.12:

$$peso_t(v) = \sum_{\substack{s \\ s \text{ é nó-filho de } t}} (reg_s(v) - mem_s(v)) + peso\,local_t(v) \tag{3.12}$$

O cálculo de $peso_t$ emprega, por sua vez, as métricas definidas nas Fórmulas 3.13 e 3.14, onde $reg?_t(v)$ retorna 1 se v já tiver sido alocado a um registrador no nó t e 0 caso contrário, enquanto $mem?_t(v)$ retorna 1 se v sofreu *spill* em t e 0 caso contrário. A função $reg_t(v)$ representa o prejuízo de se fazer *spill* de v no nó-pai de t, sendo que a variável foi alocada a um registrador em t, enquanto a função $mem_t(v)$ representa o custo de se alocar v a um registrador em um nó-pai sendo que v sofreu *spill* em t.

$$reg_t(v) = reg?_t(v)\min\{transferência_t(v), peso_t(v)\} \tag{3.13}$$

$$mem_t(v) = mem?_t(v)transferência_t(v) \tag{3.14}$$

O alocador é capaz então de combinar as heurísticas de custo a uma análise da estrutura hierárquica do programa para minimizar a inserção de *spill code*, analisando quando é proveitoso alocar uma variável para registrador ou enviá-la para memória. Por exemplo, pode ser preferível realizar *spill* de uma variável v definida em um nó t que sofreu *spill* em todas as subárvores abaixo de t, pois isso tornaria desnecessário todo o *spill code* inserido nos nós descendentes de t e a quantidade de instruções de acesso a memória seria potencialmente reduzida. Nesse caso, $peso_t(v) < 0$ indicaria um desincentivo para se alocar um registrador para v, independente de sua coloribilidade.

O principal objetivo atingido por Callahan e Koblenz era o de se obter um método de alocação que tirasse proveito do paralelismo da máquina que executa o compilador. Eles observaram aproveitamento dos recursos de paralelismo ao realizar experimentos com um compilador de Fortran [49], confirmando as expectativas prévias da dupla.

3.6 *Live Range Splitting*

Em 1998, Cooper *et al.* [6] introduziram um novo método heurístico para minimizar a inserção de código *spill*, denominado *live range splitting*. Cooper *et al.* notaram que trabalhos anteriores já haviam proposto técnicas que realizavam a quebra de *live ranges* em pedaços menores e demonstravam melhorias no *spill code* produzido [19, 26, 50]. No

entanto, essas técnicas eram pouco ajustadas e não aproveitavam de maneira eficiente as oportunidades para dividir os *live ranges* e reduzir a quantidade de acessos à memória por serem demasiado agressivas. Sendo assim, eles desenvolveram uma heurística mais precisa para efetuar a quebra dos tempos de vida em pontos estratégicos do programa e obter as melhorias desejadas.

A heurística de Cooper *et al.* usa uma estratégia mais conservadora para efetuar a separação dos tempos de vida ao somente fazê-lo como alternativa à realização de *spill* de uma variável. Dessa maneira, o alocador computaria o custo de se fazer *spill* e compará-lo-ia ao custo de *split* de uma variável candidata, escolhendo a opção menos dispendiosa. Por conseguinte, a separação de *live ranges* ocorre somente em pontos de alta pressão de registradores, onde uma variável é quebrada ao redor de outra de modo a reduzir a pressão, e o *overhead* causado pela criação de novos *live ranges* desnecessários é evitado.

Chaitin *et al.* [1] já haviam pontuado que a inserção de *spill code* não elimina completamente as interferências de uma variável, mas apenas reduz a "área de contato" ao gerar *live ranges* menores. Cooper *et al.* notaram no entanto que se um *live range* v_1 contém completamente um segundo *live range* v_2, isto é, v_1 é vivo em todos os pontos da definição até o último uso de v_2, fazer *spill* de v_2 não eliminará a interferência v_1v_2. Dessa forma, a única forma de eliminar a interferência para este caso seria efetuar *spill* de v_1 ao redor do tempo de vida de v_2, efetivamente dividindo v_1 em duas partes, como exemplificado pela Figura 21.

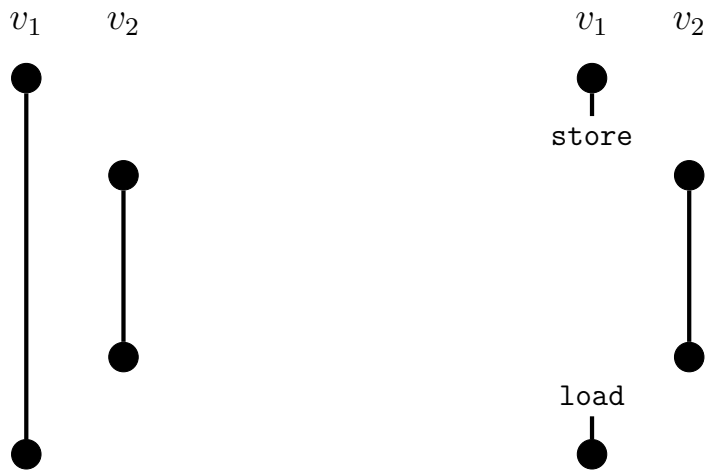


Figura 21 – Exemplo de *splitting*. Nesse caso, v_1 é dividido ao redor de v_2, eliminando a interferência.

De modo a contemplar as relações de contenção entre os *live ranges*, o *live range splitting* requer a construção de um grafo direcionado C, denominado grafo de contenção. Os nós de C representam as variáveis, enquanto uma aresta $e = (v_2, v_1)$ indica que v_1 estava vivo em uma definição ou uso de v_2. A Figura 22 exemplifica as possíveis relações entre os tempos de vida, baseado nos pontos onde são definidos e usados em relação uns aos outros.

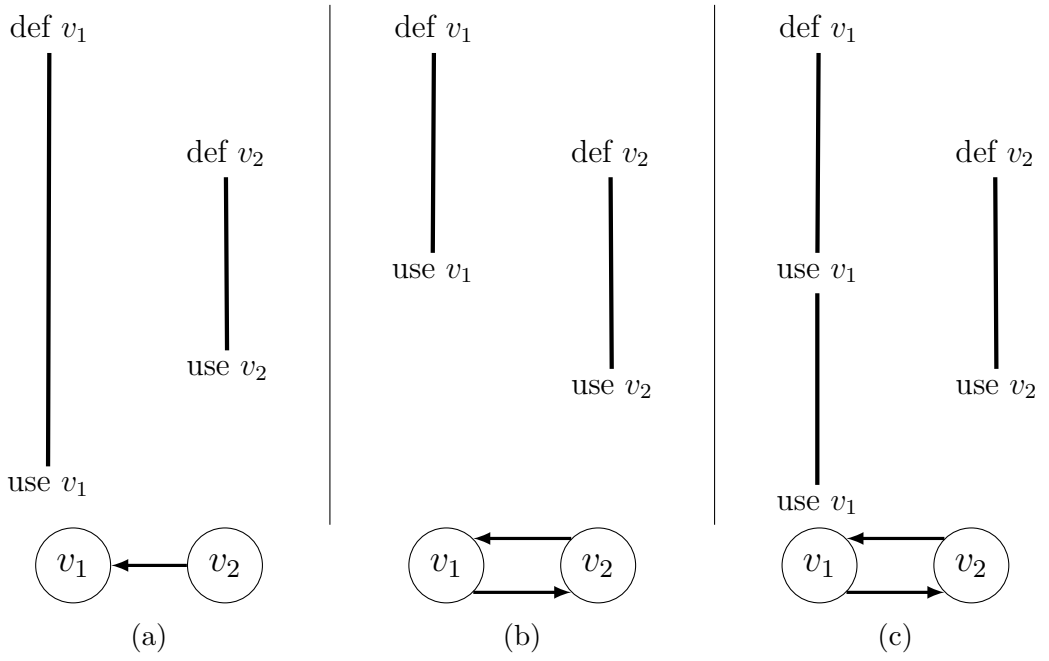


Figura 22 – Relações de contenção entre dois *live ranges* e seus respectivos grafos. Adaptado de Cooper *et al.* [6]

A coluna (a) mostra a mesma situação da Figura 21, onde v_2 não está vivo em nenhum ponto de definição ou uso de v_1 e portanto está inteiramente contido em v_1, logo $(v_2, v_1) \in C$. Em (b) ambas os *live ranges* se sobrepõe, então ambos os caminhos (v_1, v_2) e (v_2, v_1) estão em C. O caso (c) tem o mesmo resultado de (b) devido à ocorrência de um uso de $v1$ em meio ao tempo de vida de v_2. A Tabela 1 resume as possibilidades de *splitting* ou *spilling* de acordo com a ocorrência de arestas no grafo de contenção C.

Arestas em C	Efeito na geração de *spill code*
(v_i, v_j)	A divisão de v_j ao redor de v_i elimina a interferência. O *spill* de v_i, não.
(v_j, v_i)	A divisão de v_i ao redor de v_j elimina a interferência. O *spill* de v_j, não.
(v_i, v_j) e (v_j, v_i)	Não se pode dividir nenhum dos dois. Nenhum *spill* remove a interferência.

Tabela 1 – Casos de *splitting* de acordo com a ocorrência de arestas no grafo de conteção C. Adaptado de Cooper *et al.* [6].

Uma vez conhecidas as variáveis aptas a serem divididas com sucesso, o custo de fazê-lo deve ser computado pelo alocador para decidir entre *spill* ou *split*. O custo da divisão de cada *live range* v é obtido contando uma instrução `store` antes da definição e uma instrução `load` após o último uso, o que pode ser facilmente incorporado à fase de *spill costs* de um alocador ao estilo Briggs. Caso v seja escolhido para *spill* durante a fase de seleção, uma cor deve ser encontrada para v com base no custo de dividi-la em v ou o contrário. Se uma cor for encontrada, v a recebe e os tempos de vida correspondentes são marcados para divisão.

Foram realizados experimentos com 32 *benchmarks* e obteve-se resultados positivos na maioria deles. Em alguns casos, a diminuição na contagem de instruções de acesso à memória chegou a 78%. No entanto, dois casos de teste apresentaram uma maior inserção de *spill code*, e o *live range splitting* obteve piores resultados quando comparado ao *spilling* por região de interferência para mais da metade dos casos. Apesar disso, Cooper *et al.* ressaltaram a viabilidade de combinar ambas as técnicas em um alocador capaz de decidir qual delas utilizar.

3.7 *Outras Técnicas*

Coloração prioritária — Chow *et al.* publicaram em 1984 um trabalho descrevendo um algoritmo de coloração de grafos similar ao de Chaitin [1], mas com a aplicação de uma heurística de prioridade para a etapa de seleção. O alocador computa a diferença de custo entre se armazenar uma variável na memória ou e em um registrador físico e, no processo de coloração, é dada prioridade aos vértices cujo custo do *live range* associado é o mais promissor. Experimentos foram realizados em 13 *benchmarks* consistindo de programas Pascal e Fortran, e foi observada uma melhora média de 39% no tempo de execução dos binários gerados.

4 APRENDIZADO DE MÁQUINA

O aprendizado de máquina ou *"machine learning"*, em inglês, é uma área da inteligência artificial que se encarrega do desenvolvimento de agentes inteligentes que se tornam precisos ao aprenderem a partir de dados e melhorarem a si mesmos, em vez de serem explicitamente programados para tal. Esse processo de aprendizado, denominado "treinamento", comumente envolve a utilização de um conjunto de dados contendo os resultados esperados de um determinado problema, que são contrastados com os resultados produzidos pelo modelo de inteligência artificial para que o próprio modelo aprimore a si mesmo [7].

O aprendizado de máquina é particularmente útil para problemas cujos quais não há um algoritmo predefinido para resolvê-los, mas há uma grande disponibilidade de dados. Embora não seja possível encontrar uma solução exata, através da análise dos dados e apoiando-se em fundamentos da estatística e probabilidade é possível obter uma aproximação suficientemente precisa [9]. A Figura 23 ilustra o processo de desenvolvimento de um modelo genérico de *machine learning*, desde o preparo dos dados até a etapa de avaliação da performance do agente inteligente.

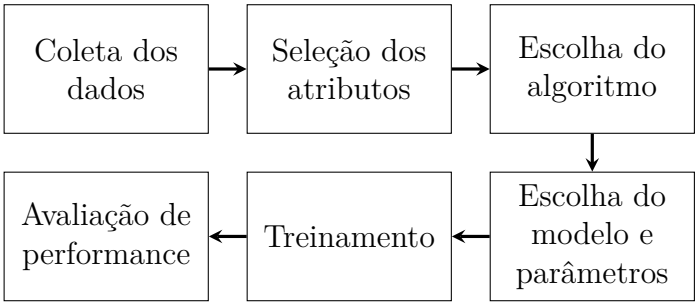


Figura 23 – Esquema ilustrando o desenvolvimento de um modelo genérico de *machine learning*. Adaptado de Alzubi *et al.* [7].

Sendo assim, as soluções do aprendizado de máquina consistem da identificação de padrões regulares na natureza do problema e na utilização desses padrões para realizar predições. Dentre as tarefas mais comuns dos modelos de *machine learning* encontram-se [7]:

- **Classificação** — classificar os dados de entrada em um conjunto fixo de classes de saída conhecidos antecipadamente como sim ou não, verdadeiro ou falso, dentre outras;
- **Detecção de anomalias** — buscar padrões nos dados de entrada de modo a identificar desvios ou anomalias. Exemplos reais incluem análise de tráfego de rede e identificação de transações fraudulentas;

- **Regressão** — encontrar uma função matemática que descreva o resultado para uma dada entrada, com base nos dados disponíveis para treinamento. Exemplos são a estimativa de tendências de crescimento de preços ou crescimento populacional;
- **Agrupamento** — encontrar estruturas nos dados e separá-los em grupos. Um exemplo real é a análise de texto para agrupá-los por classe de documentos semelhantes.

Com o avanço da capacidade de armazenamento e das tecnologias de redes nas últimas décadas, tornou-se possível armazenar e processar grandes quantidades de dados, bem como acessá-los remotamente por meio da internet. Isso viabilizou o amplo emprego das técnicas de aprendizado de máquina em uma série de aplicações [9, 27], incluindo a criação de compiladores e a geração de *spill code*, o que será abordado no Capítulo 5 deste trabalho. O restante do atual Capítulo irá sumarizar a fundamentação teórica por trás dos principais paradigmas de aprendizado e técnicas de implementação de modelos de *machine learning*.

4.1 Aprendizado Supervisionado

No paradigma de aprendizado supervisionado, cada entrada do conjunto de dados de treinamento é rotulada de acordo com as variáveis relevantes ao processo de aprendizagem. Em termos formais, cada elemento do conjunto de treinamento é definido como composto pelo vetor das variáveis de entrada $\vec{X}$ e um rótulo r, tal que o conjunto de treinamento seja caracterizado por um mapeamento $m : \vec{X} \rightarrow \{r_1, r_2, \ldots, r_n\}$. O processo de treinamento então visa ajustar o modelo para produzir um mapeamento $m' : \vec{X} \rightarrow \{r_1, r_2, \ldots, r_n\}$, onde m' é suficientemente próximo da relação m original presente no conjunto de treinamento [15, 9].

Neste contexto, os rótulos para o vetor de saída são fornecidos por um supervisor, que pode ser tanto um humano quanto uma máquina. Embora a rotulagem por humanos seja mais cara e demorada, os erros mais frequentes na rotulagem feita por máquinas sugerem a superioridade do julgamento humano. Dados rotulados manualmente são considerados recursos valiosos e confiáveis para o aprendizado supervisionado, embora em alguns casos, máquinas também possam ser usadas para rotulagem confiável [15].

Na Figura 24, um conjunto de treinamento contendo 4 elementos, cada qual formado por um vetor de entrada $\vec{X} = [x_1, x_2]^T$ e rotulado como $\{r_1, r_2\}$, forma um espaço bidimensional. Após um treinamento supervisionado um modelo inteligente encontra a classificação C, que compreende os elementos do conjunto que possuem o rótulo r_1. A Tabela 2 demonstra possíveis rotulações para conjuntos de dados não-rotulados, a serem atribuídos por um agente supervisor.

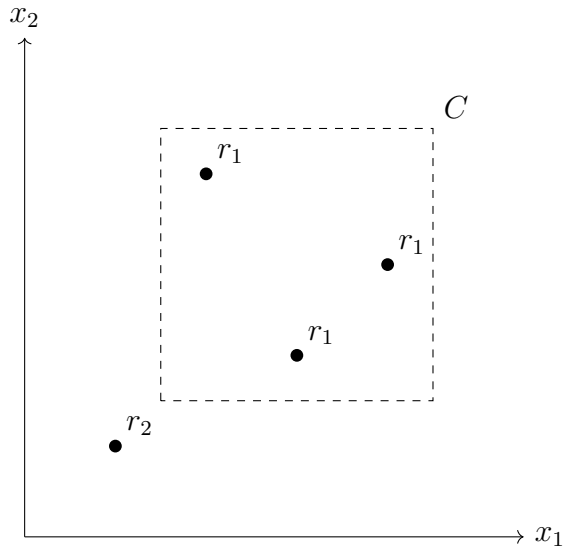


Figura 24 – Análise gráfica considerando um conjunto de treinamento com 4 elementos, formados pelos parâmetros de entrada $\vec{X} = [x_1, x_2]^T$ e rótulos $\{r_1, r_2\}$.

Exemplo de dado	Critério de rotulação	Possíveis rótulos
Tweet	Sentimento do *tweet*	{positivo, negativo}
Imagem	Contém uma casa ou carro?	{sim, não}
Áudio	A palavra "futebol" é dita	{sim, não}
Vídeo	Armas são mostradas no vídeo?	{violento, não-violento}
Raio X	Presença de tumor no raio X	{presente, ausente}

Tabela 2 – Exemplos de dados não-rotulados e possíveis critérios de rotulação a um possível supervisor. Extraído de Mohammed *et al.* [15]

4.2 Aprendizado Não-Supervisionado

No paradigma não-supervisionado, não há um supervisor para rotular os elementos do conjunto de entrada. Essa abordagem foca em reconhecer padrões não identificados previamente nos dados para derivar os critérios de classificação ou regressão a partir deles. Essa técnica é adequada para processar grandes volumes de dados onde a rotulação é demasiado trabalhosa ou impossível, realizando análises estatísticas para descobrir estruturas ocultas em dados não-rotulados [15].

Dentre os principais métodos de aprendizado não-supervisionado destacam-se o agrupamento (*clustering*) e a redução de dimensionalidade. O agrupamento envolve a tarefa de identificar grupos ou *clusters* de dados semelhantes com base em suas características, permitindo a segmentação de dados em categorias não conhecidas previamente. A redução de dimensionalidade busca representar dados complexos em um espaço de menor dimensão, preservando as características importantes, o que é útil para simplificar a análise de dados de alta dimensionalidade e visualização [51].

Algoritmos populares incluem o *k-means* para agrupamento, que divide os dados

em *clusters*, ou agrupamentos, com base na proximidade dos pontos de dados uns com os outros espaço n-dimensional [9, 15], e a análise de componentes principais (PCA) para redução de dimensionalidade, que é uma técnica que reduz a dimensionalidade dos dados, mantendo as informações mais importantes e descartando as menos importantes [8].

A Figura 26 mostra dados plotados em 2 dimensões divididos em 2 *clusters*, cujos centroides [1] são indicados por um "×". A Figura 25 mostra uma visualização de um conjunto de dados de *microarray* sobre câncer de mama usando mapas elásticos. O *plot* (a) mostra nós projetados em um espaço tridimensional; o conjunto de dados é curvo e não pode ser adequadamente mapeado em um plano principal bidimensional. O *plot* (b) mostra a distribuição nas coordenadas internas da superfície principal não linear bidimensional e (c) o mesmo que (b), mas para o espaço linear bidimensional após realização do PCA.

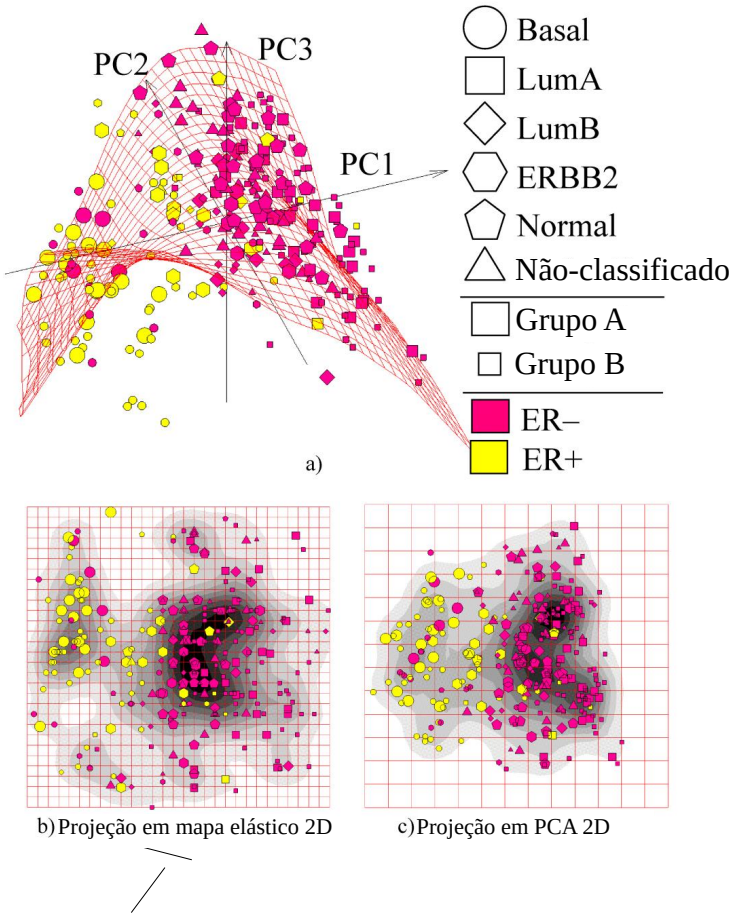


Figura 25 – *Plot* de um conjunto de dados de *microarray* de câncer de mama usando mapas elásticos, empregando técnicas de análise de componentes principais. Extraído de Gorban e Zinovyev [8].

[1] O centroide corresponde ao ponto médio de todos os elementos de um determinado agrupamento no espaço k-dimensional.

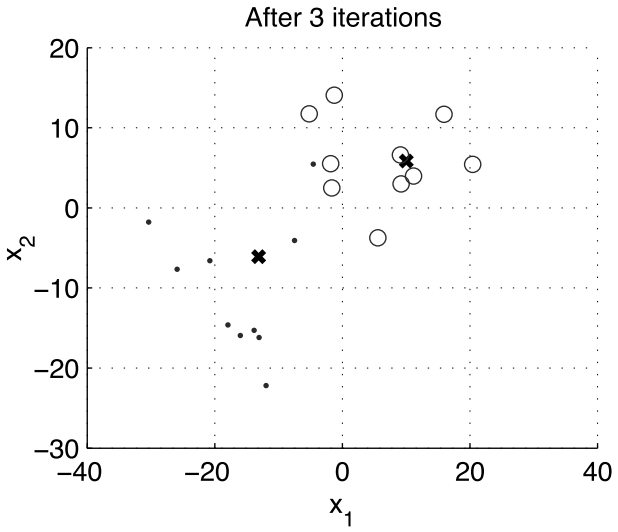


Figura 26 – Exemplo de aplicação do *k-means*. Os centroides dos *clusters* são indicados com um "×" no gráfico. Extraído de Alpaydin [9].

4.3 Aprendizado Semi-supervisionado

No aprendizado semi-supervisionado os dados fornecidos consistem em uma mistura de dados rotulados e não rotulados. Essa combinação de ambos os tipos é usada para criar um modelo apropriado para a classificação de dados sendo que, frequentemente, os dados rotulados são escassos, enquanto os não rotulados são abundantes. O objetivo da classificação semi-supervisionada é aprender um modelo que preveja as classes de dados de teste futuros melhor do que um modelo gerado apenas com os dados rotulados. Isso permite aproveitar ao máximo os dados não rotulados disponíveis para melhorar o desempenho da classificação [15].

Alguns dos principais métodos de treinamento empregados no aprendizado semi-supervisionado incluem, dentre outros [52, 53]:

1. **Modelos generativos** — visam aprender a distribuição de probabilidade dos dados não rotulados com base nos dados rotulados conhecidos. Eles permitem não apenas fazer previsões ou classificações, mas também gerar novos dados que se assemelham aos dados de treinamento. Esses modelos são úteis para a geração de dados sintéticos e para melhorar o desempenho da classificação;

2. ***Self-training*** — um agente classificador é treinado usando os dados rotulados, e em seguida é usado para rotular o restante dos dados não classificados do conjunto de treinamento;

3. ***Co-training*** — se baseia na ideia de usar múltiplas visualizações ou conjuntos de características dos dados para treinar modelos independentes. Essa abordagem é particularmente útil quando os dados disponíveis podem ser divididos em diferentes visualizações que fornecem informações complementares.

4.4 Aprendizado por Reforço

No aprendizado por reforço, o agente inteligente toma decisões em um ambiente e recebe recompensas (ou penalidades) por suas ações ao tentar resolver um problema e, após uma série de tentativas e erros, o agente deve apresentar a melhor política de tomada de decisões, que é a sequência de ações que maximiza a recompensa total [9]. A política é dada pelo mapeamento $\pi : A \times S \rightarrow [0, 1]$, onde A é o conjunto das ações e S é o conjunto dos estados de *input*. A política $\pi(a, s)$ retorna a probabilidade da ação $a \in A$ ser tomada dado o estado $s \in S$ do ambiente [54].

O treinamento se dá de maneira iterativa, quando o agente observa o estado de entrada no ambiente e, em seguida, utiliza uma função de tomada de decisão para escolher e realizar uma ação específica. Após a execução da ação, o agente recebe uma recompensa ou reforço do ambiente, que pode ser positiva (indicando uma ação benéfica) ou negativa (indicando uma ação prejudicial). A geração de políticas de tomada de decisão pode ser feita através de força-bruta ou empregando heurísticas fundamentadas em processos estocásticos para a escolha da próxima ação, se baseando nas ações prévias do sistema e o atual estado [15, 54], sendo que esse processo é ilustrado pela Figura 27.

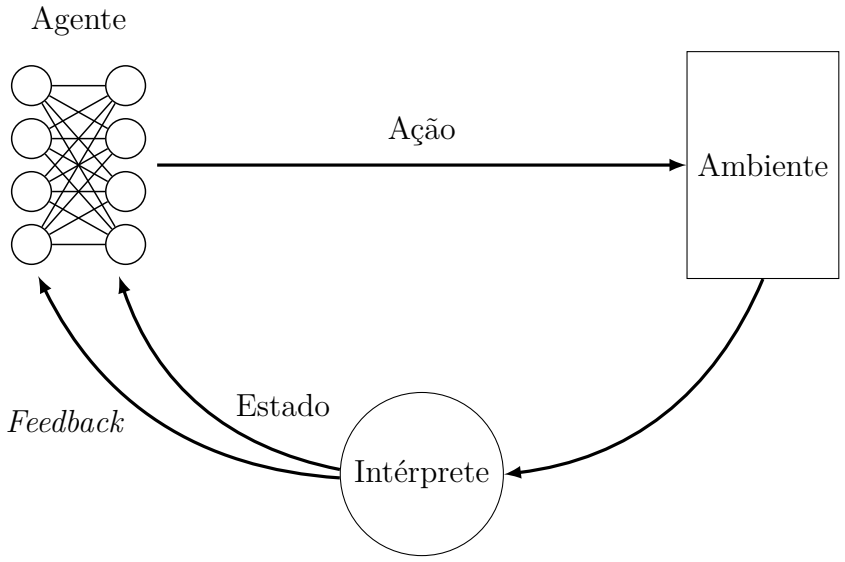


Figura 27 – Esquema do processo de treinamento via aprendizado por reforço.

O aprendizado por reforço difere dos outros paradigmas de aprendizado de várias maneiras; a diferença mais importante é a não utilização de pares de entrada/saída esperada, como no aprendizado supervisionado. Em vez disso, depois de escolher uma ação, o agente é informado sobre a recompensa imediata e o estado subsequente, mas não é informado sobre qual ação teria sido do seu melhor interesse a longo prazo. É necessário para o agente adquirir experiência útil sobre os possíveis estados do sistema, ações, transições e recompensas ativamente para agir de forma otimizada. Outra diferença em relação ao aprendizado supervisionado é que o desempenho em tempo real é importante: a avaliação

do sistema muitas vezes ocorre simultaneamente ao aprendizado [54].

4.5 Redes Neurais Artificiais e Aprendizado Profundo

As redes neurais artificiais (ANNs) constituem uma classe de modelos de inteligência artificial que têm como inspiração para seu funcionamento o sistema nervoso dos animais. As redes neurais são compostas de uma série de unidades chamadas neurônios, que funcionam como pequenos computadores de função individuais; cada neurônio artificial recebe uma ou mais entradas, computa uma função matemática tendo como parâmetros as entradas recebidas e produz uma saída, que é propagada para a camada seguinte. Uma rede neural possui ao menos uma camada de entrada e uma camada de saída, que apresenta o resultado [55, 56].

As redes neurais que resolvem problemas de classificação são chamadas de "*percéptrons*", ao dividir o espaço dos valores de entrada com uma série de hiperplanos[2], onde os valores de cada lado dos hiperplanos pertencem a classes distintas. Problemas onde é necessário o uso de somente um hiperplano são ditos linearmente separáveis, e podem ser resolvidos por redes neurais simples. Entretanto, grande parte dos problemas reais não são linearmente separáveis. Nesses casos, é necessário a utilização de mais hiperplanos através da introdução de camadas ocultas na rede neural que realizam, cada uma, classificações parciais que são repassadas à camada seguinte, até que se chegue na camada final [55, 9].

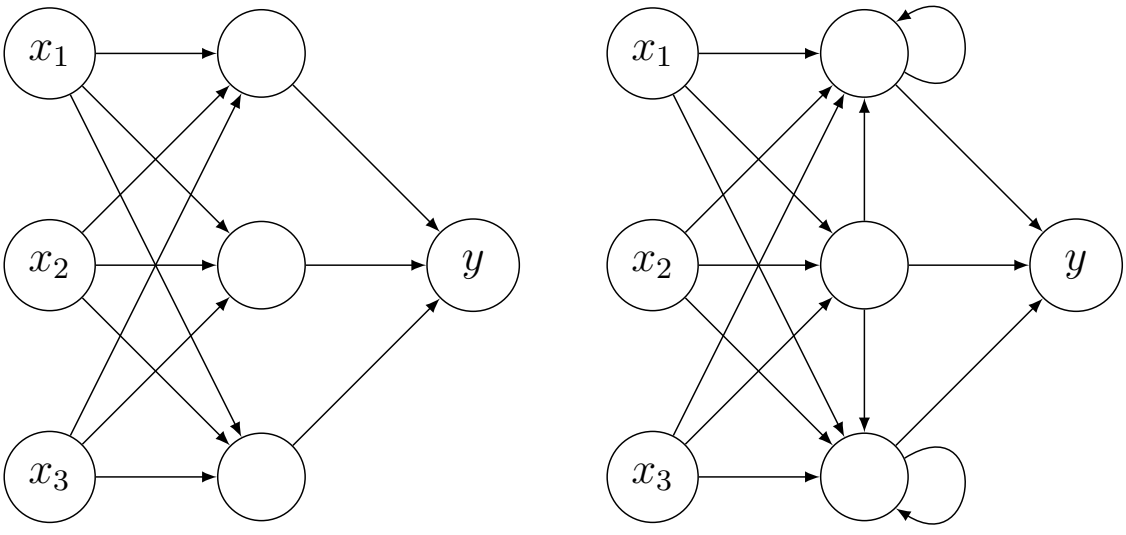


Figura 28 – Uma rede neural *feed-forward* e uma rede neural recorrente, na esquerda e direita respectivamente.

[2] Generalização do conceito de plano para dimensões maiores. Dado um espaço k-dimensional, um hiperplano é um subespaço $k-1$-dimensional. O correspondente a um hiperplano em um espaço bidimensional é uma reta; em um espaço tridimensional, é um plano propriamente dito.

Quando uma rede neural possui uma ou mais camadas intermediárias entre a entrada e a saída, ela é denominada "rede neural profunda", e seu processo de treinamento é chamado de aprendizado profundo, ou *"deep learning"*. Como mostrado na Figura 28, as redes neurais profundas podem ter uma organização *"feed-forward"*, forma mais amplamente utilizada e que apresenta um fluxo unidirecional de uma camada para outra, ou recorrente, que pode conter ciclos e possuir um fluxo bidirecional entre as camadas [56].

A estrutura básica de um neurônio artificial é mostrada na Figura 29. Ele recebe uma ou mais entradas x_1, x_2, ..., x_n, cada qual associada a um peso dentre w_1, w_2, ..., w_n, e realiza uma soma ponderada para produzir uma saída, também chamada de ativação. Em certos casos a soma é adicionada a um termo b conhecido como viés ou limiar, antes de passar por uma função f não linear chamada função de ativação ou função de transferência.

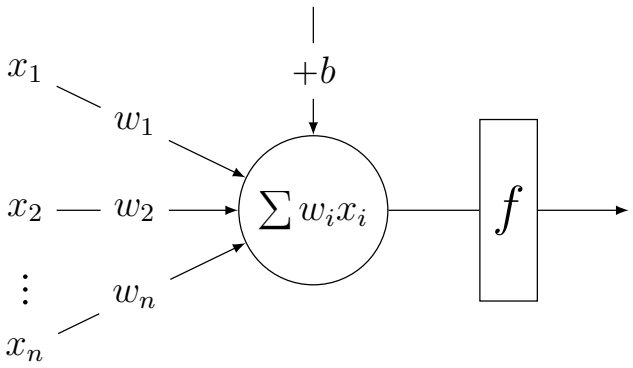


Figura 29 – Esquema do funcionamento básico de um neurônio artificial.

As funções de transferência geralmente possuem uma forma sigmoide, mas também podem adotar a forma de outras funções não-lineares, funções lineares em partes ou funções degrau. Algumas importantes funções usadas como limiar incluem [57]:

- ***Função degrau*** — saída binária, se a entrada for maior que um limite especificado θ:

$$f(x) = \begin{cases} 1, & x \geqslant \theta \\ 0, & \text{caso contrário.} \end{cases}$$

- ***Função logística*** — função sigmoide (curva em forma de "S"), utilizada para gerar um valor no intervalo $[0, 1]$:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- ***Tangente hiperbólica*** — função sigmoide com domínio limitado no intervalo $[-1, 1]$, sendo simétrica em torno de 0:

$$f(x) = \tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- ***Rectified Linear Unit* (ReLU)** — retorna a entrada se for positiva e zero caso contrário. É uma função de ativação popular em redes neurais profundas devido à eficiência no processo de treinamento:

$$f(x) = \max\{0, x\}$$

O treinamento dos modelos de rede neural pode ser feito através de um série de algoritmos, dentre os quais se destaca o método do *back-propagation*, ou retropropagação, que faz uso de uma técnica de otimização numérica chamada descida gradiente para o treinamento de redes profundas *feed-forward*. A rede é inicializada com todos os seus pesos recebendo valores aleatórios e a primeira iteração é realizada, onde uma entrada é analisada e é produzida uma saída aleatória. Em seguida, é calculado um erro, igual ao quadrado da diferença entre o último resultado e o resultado esperado, e os parâmetros da rede (pesos e limiares) são ajustados de modo a reduzir o erro calculado. Esse processo é repetido inúmeras vezes, até que se obtenha o conjunto de pesos que minimizam o erro, e produzam um resultado muito próximo do esperado [55].

Redes neurais profundas, incluindo as *feed-forward* e as recorrentes, desempenham um papel fundamental em uma ampla variedade de aplicações. Na visão computacional, elas são usadas para tarefas como detecção de objetos, reconhecimento facial e classificação de imagens. No processamento de linguagem natural, auxiliam na análise de sentimento, tradução automática, geração de texto e reconhecimento de fala. Já as redes neurais recorrentes destacam-se no processamento de sequências, como tradução automática e geração de texto coerente, bem como na modelagem de linguagem. A versatilidade do *deep learning* e a capacidade de aprender padrões complexos apresentada pelas redes neurais profundas as tornam essenciais em muitos campos, e representam um grande horizonte a ser investigado nas pesquisas [27].

5 TRABALHOS CORRELATOS

Com a recente expansão da pesquisa em torno do aprendizado de máquina, uma variedade de trabalhos foram realizados buscando integrar as soluções de *machine learning* à área de compiladores em geral e, consequentemente, aos problemas da alocação de registradores e geração de código *spill*. Todavia, alguns obstáculos se fizeram presentes, como a indisponibilidade de *datasets* para treinamento e a necessidade por corretude nos algoritmos de geração de código, contrastando com a natureza aproximada das soluções produzidas por modelos de *machine learning* [13].

Ainda assim, uma variedade de trabalhos apresentam resultados promissores ao utilizar técnicas de aprendizado de máquina para a obtenção de melhores heurísticas e promover uma minimização de *spill code* mais eficiente. Este Capítulo tem por objetivo expor alguns desses trabalhos e apresentar um panorama do estado da arte das pesquisas recentes combinando aprendizado e máquina e alocação de registradores.

5.1 *Meta Optimization*

Em 2003, Stephenson *et al.* [10] publicaram um trabalho que discute o uso de técnicas de aprendizado de máquina para ajustar automaticamente as heurísticas de várias etapas de otimização, incluindo a alocação de registradores. Dessa forma, os autores introduzem o conceito de *"Meta Optimization"* ou metaotimização, em uma tradução livre, uma metodologia que emprega *machine learning* e auxilia os desenvolvedores de compiladores no trabalhoso processo de ajuste fino das heurísticas que norteiam as de otimização de código.

Após analisar várias otimizações de compiladores, os autores constataram que muitas heurísticas têm um ponto focal: uma única função de prioridade ou custo que muitas vezes dita a eficácia de uma heurística. Uma função de prioridade leva em conta os fatores que afetam um determinado problema, e apresenta uma medida da importância relativa das opções disponíveis em um processo de otimização. Sendo assim, o *Meta Optimization* consiste de um modelo de aprendizado que envolve programação genética para vasculhar, de maneira iterativa, o conjunto de soluções para a função de prioridade de uma otimização, em busca da alternativa que produza os melhores resultados.

A programação genética (GP) é um tipo de algoritmo evolutivo, isto é, um paradigma de inteligência artificial que se inspira na evolução das espécies por seleção natural, proposta por Charles Darwin [58]. Na programação genética, os possíveis programas ou rotinas que resolvem um determinado problema são representados de maneira abstrata como indivíduos de uma população, definidos por um "genoma" que pode ser recombi-

nado ou alterado de modo a criar novos indivíduos. O processo de obtenção da solução ótima consiste em, ao longo de várias iterações ou gerações, criar variabilidade na população através de reprodução cruzada, mutações ou recombinações, e em seguida efetuar a seleção dos indivíduos mais aptos, criados aleatoriamente [59].

Na abordagem de Stephenson *et al.* [10], as possíveis heurísticas são submetidas ao processo de seleção, sendo cada uma representada como a árvore sintática abstrata da expressão que corresponde ao cálculo da função. O algoritmo começa criando uma população composta de expressões iniciais usadas como referência e outras expressões geradas aleatoriamente. Em seguida, o algoritmo determina o nível de *fitness*, ou aptidão, de cada indivíduo, nesse caso correspondente à velocidade de execução do código gerado utilizando a função de prioridade na alocação de registradores, e é dado início ao processo de evolução. A Figura 30 ilustra, nas Subfiguras (a) e (b), duas árvores sintáticas que correspondem a duas funções de prioridade; a Subfigura (c) mostra o resultado do cruzamento dos genomas de (a) e (b), enquanto (d) apresenta uma mutação na Subfigura (a).

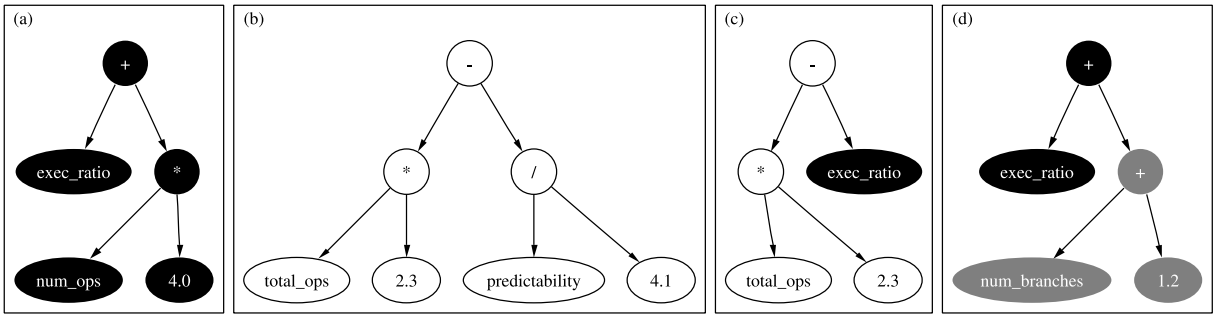


Figura 30 – Genomas das funções de prioridade, submetidas a operações de cruzamento e mutação. Extraído de Stephenson *et al.* [10].

O algoritmo de alocação de registradores usado nos experimentos foi o de alocação prioritária, utilizando uma função de prioridade que representa o ganho de performance ao se manter um *live range* em registradores, ao invés de se realizar *spill*. A Equação 5.1 mede o ganho de performance ao se manter uma variável em registradores no bloco i, onde $ganho_{load}$ e $ganho_{store}$ representam as economias ao se evitar *loads* e *stores* respectivamente; $uses_i$ e $defs_i$ são os números de usos e definições de uma variável no bloco i, enquanto w_i é a frequência de execução de i. A Equação 5.2 define a função prioritária de referência que foi utilizada para inicializar a população de treinamento, onde V é um *live range* e N é o número de blocos que compõe V.

$$ganhos_i = w_i(ganho_{load} \times uses_i + ganho_{store} \times defs_i) \tag{5.1}$$

$$prioridade(V) = \frac{\sum_{i \in V} ganhos_i}{N} \tag{5.2}$$

Os experimentos realizados por Stephenson *et al.* apresentaram um aumento médio de 11% na velocidade de execução com as heurísticas especializadas obtidas pelo *Meta Optimization*, isto é, treinando uma população para cada *benchmark* de teste. Em experimentos buscando uma heurística de uso geral, utilizando diferentes *benchmarks* para a mesma população, o aumento médio foi de 3%, que, apesar de menor do que o obtido com as heurísticas especializadas, ainda se mostra um aumento significativo. A Figura 31 mostra o *plot* do nível de *fitness* ao longo das gerações, para o treinamento das heurísticas especializadas e da heurística de uso geral, respectivamente.

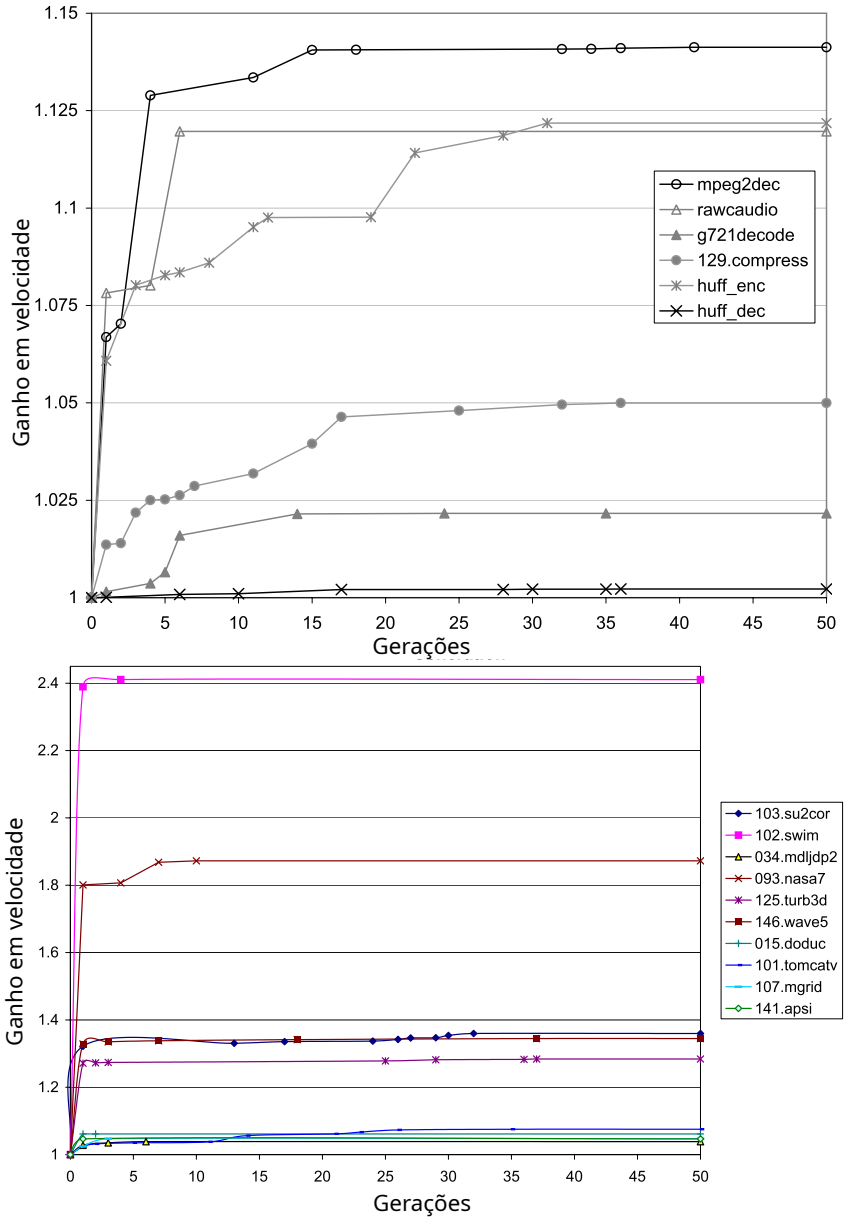


Figura 31 – O eixo horizontal mostra o número de gerações, enquanto o eixo vertical apresenta o aumento na velocidade de execução. Extraído de Stephenson *et al* [10].

5.2 Coloração de Grafos via *Deep Learning*

Em um trabalho de 2019, Lemos *et al.* [60] lidam com problema da coloração utilizando um modelo de rede neural em grafo (GNN), um tipo de rede neural capaz de representar informações extraídas dos nós de um grafo [61]. O modelo utiliza um sistema de memória que mantém representações multidimensionais das informações sobre os vértices, associados a um conjunto de cores, além de uma rede neural recorrente (RNN) que computa as atualizações no sistema. O processo envolve 32 iterações de troca de mensagens entre vértices adjacentes e entre vértices e cores, para que o modelo obtenha a viabilidade das colorações para cada vértice. Cada vértice então decide se o grafo de entrada permite uma coloração com k cores.

O modelo foi treinado utilizando uma abordagem generativa. As instâncias de treinamento foram criadas tomando instâncias reais e modificando seu número cromático ao adicionar vértices no grafo; em seguida, ambas as instâncias eram adicionadas no conjunto de treinamento. O modelo atingiu 82% de precisão para instâncias após entre 40 e 60 gerações de treinamento, e número de cores k entre 3 e 7. Além disso, os autores observaram que o modelo treinado generaliza bem para valores de k não vistos anteriormente e instâncias maiores.

Os pesquisadores discutem na publicação como o funcionamento interno do modelo influencia sua tomada de decisão. Eles afirmam que o modelo procura uma resposta positiva ao agrupar vértices de forma a aproximar aqueles que poderiam ter a mesma cor. Apesar de agrupar vértices adjacentes em uma proporção baixa, o modelo continua a fornecer respostas positivas. O número desejado de cores (k) é introduzido no modelo com representações iniciais aleatórias, evitando qualquer conhecimento prévio.

Eles propõem melhorias futuras, como a redução de conflitos dentro do próprio modelo como uma métrica de perda, mesmo com o aumento da complexidade temporal e espacial. Os autores destacam que seu trabalho evidencia como um modelo semelhante ao GNN pode ser ajustado para solucionar desafiadores problemas combinatórios, como a coloração de grafos, de maneira interpretável, gerando resultados precisos e construtivos.

Em 2020, Das *et al.* [12] publicaram um trabalho que também envolve a alocação via coloração de grafos empregando modelos de *deep learning*, especificamente, o alocador utiliza uma abordagem híbrida, como os próprios autores assim o descreveram, dividida em duas etapas: uma consiste na coloração propriamente dita, feita por uma rede neural multicamadas, e a outra consiste de uma etapa de correção. Os autores buscaram encontrar novas heurísticas através do *deep learning*, apresentando resultados comparáveis ou até melhores do que os dos alocadores convencionais por coloração de grafos.

A coloração do grafo de interferência é feita por uma rede neural recorrente (RNN) composta por várias camadas de LSTM, ou *"long short-term memory"*. O LSTM é uma

arquitetura de RNN que possui memória, sendo capaz de armazenar valores em intervalos arbitrários. Uma célula LSTM é formada por um conjunto de neurônios organizados em "portas" que controlam o fluxo de informação: uma porta *input* para entrada, uma porta *output* para saída e a porta *forget*, que controla o esquecimento da informação. A Figura 32 ilustra a organização básica de uma célula LSTM.

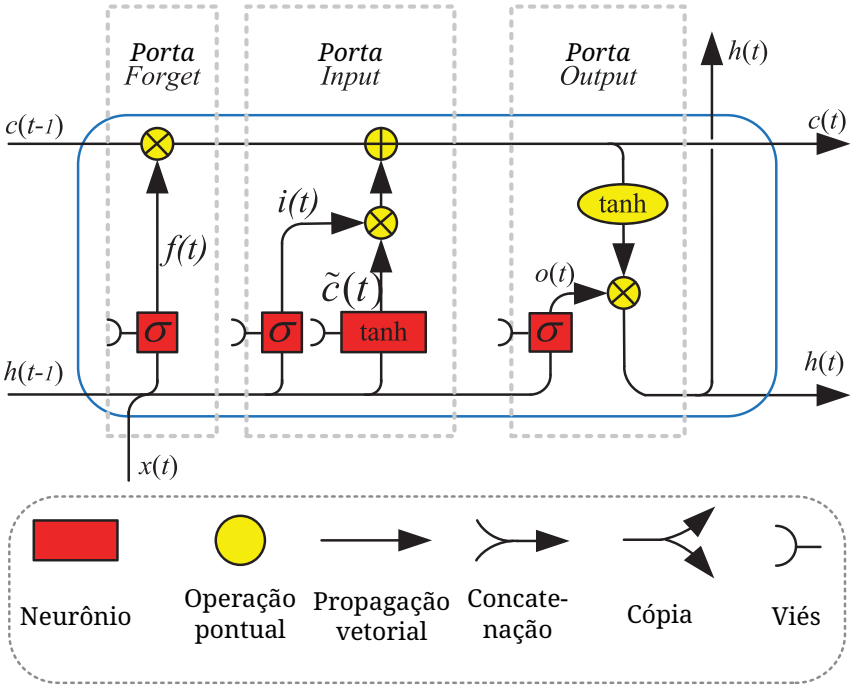


Figura 32 – Extraído de Yu *et al.* [11].

A porta *input* decide quais partes das novas informações devem ser armazenadas no estado atual, usando um sistema semelhante ao da porta *forget*. A porta *output* controla qual parte das informações no estado atual devem ser usadas, atribuindo um valor de 0 a 1 à informação, considerando os estados anterior e atual. A porta *forget* determina quais informações descartar do estado anterior, atribuindo um valor entre 0 e 1 ao estado anterior em comparação com a entrada atual. Um valor próximo a 1 significa manter a informação, enquanto um valor próximo a 0 significa descartá-la. A produção seletiva de informações relevantes do estado atual permite que a rede LSTM mantenha dependências úteis de longo prazo para fazer previsões, tanto nos passos de tempo atuais quanto nos futuros [11, 12].

Visando maximizar a performance, o modelo foi arquitetado possuindo três camadas de células LSTM, com os estados ocultos de uma camada sendo repassados às células da camada seguinte. Os dados submetidos à camada de entrada da rede neural são as adjacências de cada vértice do grafo, sendo uma lista de adjacências $adj(v_i)$ para cada vértice v_i, e é produzido como saída uma sequência de cores $cor(v_i)$, cada qual correspondente à coloração de um vértice do grafo de interferência. A Figura 33 esquematiza a organização do modelo.

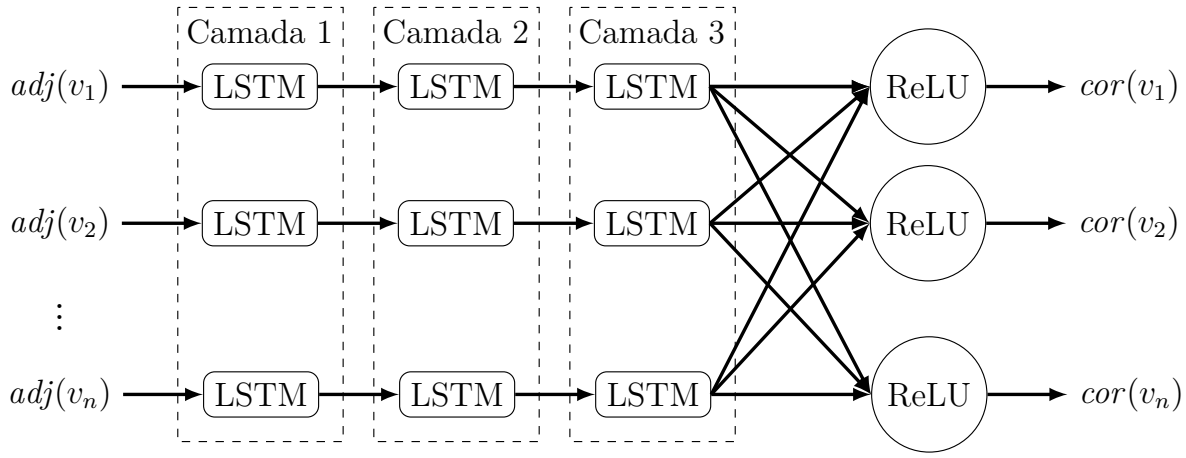


Figura 33 – Esquema da rede neural de coloração. Adaptado de Das *et al.* [12].

Todavia, em alguns casos o modelo realiza colorações inválidas, onde dois vértices adjacentes possuem a mesma cor. Sendo assim, a coloração produzida como resultado pela rede neural é submetida a uma fase subsequente de correções, onde as colorações inválidas são corrigidas. Para o treinamento, foram utilizados grafos de até 100 vértices gerados aleatoriamente usando a biblioteca `very nauty`, disponível em linguagem C. O modelo foi validado utilizando os *benchmarks* da suíte do SPEC CPU 2017, e o desempenho do alocador foi comparado ao do alocador `greedy` do LLVM, um *framework* em código aberto que contém várias ferramentas para o desenvolvimento de compiladores [62].

O desempenho foi medido em relação ao número de registradores necessários para obter uma coloração válida sem necessidade de *spill*. Sendo assim, o desempenho foi medido utilizando cinco *benchmarks*. Em um deles (`508.namd_r`), o modelo de *deep learning* apresentou um resultado pior do que o alocador do LLVM, sendo 5% pior. Nos demais (`505.mcf_r`, `557.xz_r`, `541.leela_r` e `502.gcc_r`), a rede neural apresentou um desempenho superior, de 2%, 2,5%, 7% e 5%, respectivamente.

5.3 Alocação de Registradores com *Reinforcement Learning*

Em 2023, VenktaKeerthy *et al.* [13] apresentaram uma aplicação que realiza a alocação de registradores empregando técnicas de *reinforcemente learning* hierárquico e as ferramentas do LLVM. Eles abstraíram o problema de alocação dividindo-o em uma série de tarefas, cada uma sendo realizada por um agente inteligente, que se integram maneira hierárquica ao gerador de código do LLVM através de um *framework* gRPC. A aplicação realiza a coloração de grafos de maneira não-iterativa, também sendo capaz de efetuar *live range splitting*.

De maneira semelhante ao modelo de coloração de Lemos *et al.* [60] (5.2), o modelo de inteligência artificial de VenktaKeerthy *et al.* [13] representa o grafo de interferência codificando as instruções da representação intermediária de máquina (MIR) do LLVM,

que então compõe o *input* de uma *gated graph neural network* (GGNN). A GGNN mantém uma visão do estado grafo de interferência, e a mantém atualizada conforme as alocações são realizadas realizando troca constante de informações entre os vértices. Além disso, elas permitem a anotação dos nós e arestas com base em seus tipos e propriedades, levando em consideração essas informações durante o aprendizado das representações.

O alocador foi denominado *RL4ReAl* é composto de quatro agentes que se encarregam, cada um, de uma tarefa específica do processo de alocação de registradores. São eles, em ordem hierárquica:

1. ***Node selector*** — agente encarregado de selecionar um vértice $v \in G$ levando em conta o custo de *spill* e que, por consequência, determina a ordem de alocação. A política aprendida é considerada boa com base no resultado final da alocação. Portanto, a recompensa para este agente também é modelada com base nas recompensas dos agentes de nível inferior;

2. ***Task selector*** — agente encarregado de decidir se uma variável específica será alocada para um registrador ou passará pelo processo de *live range splitting*, levando em conta o número de registradores disponíveis, o número de interferências, seu tempo de vida e o custo de *spill*. Sua recompensa é determinada com base na coloração da variável. Se a tarefa escolhida for o *splitting*, então a recompensa é adiada até a decisão de coloração.

3. ***Splitter*** — agente responsável por determinar os pontos de divisão dos *live ranges* eleitos para *splitting*. Para prever onde dividir os *live ranges*, são levados em conta os custos de *spil* em cada uso da variável, as distâncias entre usos sucessivos da variável e a codificação do vértice correspondente na GGNN. A recompensa é dada pela variação nas distâncias dos usos sucessivos da variável que sofreu *split* e pela variação do número de interferências no grafo. Se a variação das distâncias de uso for maior do que a das interferências, a divisão é benéfica.

4. ***Coloring agent*** — agente de nível inferior que aprende a selecionar uma cor válida para uma variável ou realizar *spill* caso não haja registrador disponível. Sua recompensa é determinada pelo custo de *spill* da variável: se o tempo de vida for colorido, o reforço é positivo; no entanto, se a variável for mapeada para memória, o reforço é negativo. Dessa forma, o agente aprende a priorizar a coloração de variáveis com custo mais elevado.

Esses agentes constituem o modelo de *reinforcement learning* implementado em linguagem Python, que interage com o otimizador do LLVM através de um *framework* gRPC, que permite a realização de chamadas remotas entre programas diferentes. Dessa maneira, informações do grafo de interferência construído com as ferramentas do LLVM

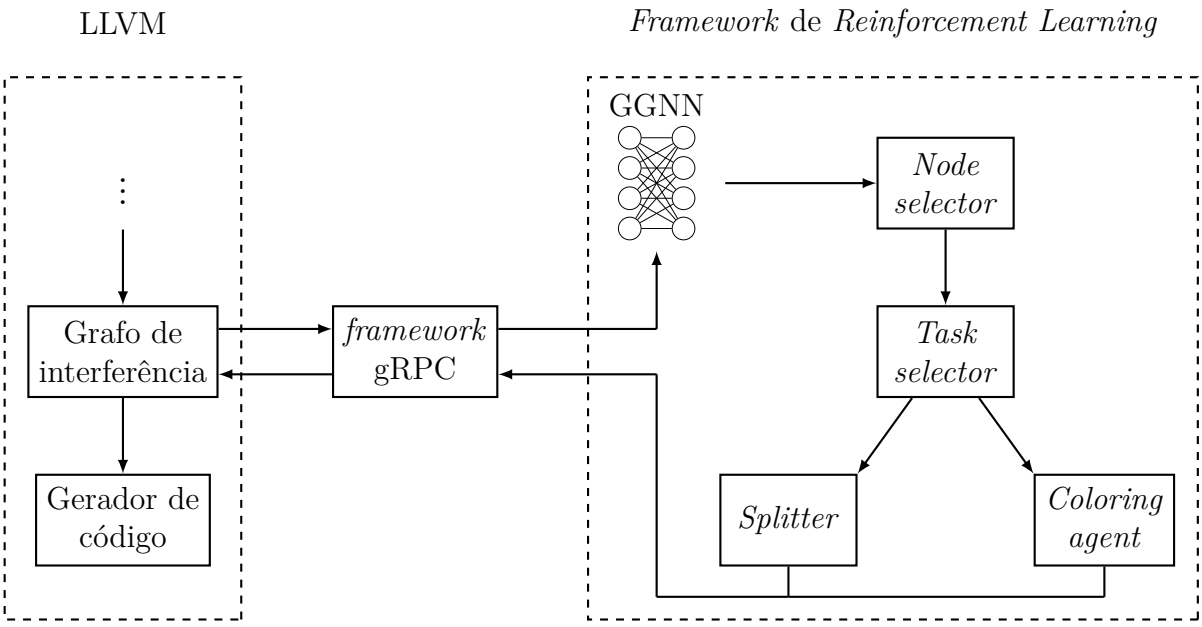


Figura 34 – Esquema da interação entre o LLVM e o *framework* de *RL*. Adaptado de VenkataKeerthy *et al.* [13].

são enviadas para os agentes, e as decisões tomadas pelo *Splitter* ou pelo *Coloring agent* são devolvidas de modo a atualizar a representação do grafo presente no LLVM. A Figura 34 esquematiza a interação entre ambos os componentes através do gRPC.

A tomada de decisões por parte dos agentes foi modelada como um processo de decisão de Markov, que representa processos de decisão sequenciais estocásticos em um sistema de estados, onde funções de custo e transição dependem apenas do estado atual do sistema e da ação atual [14]. A Figura 35 mostra um sistema de decisão de Markov, onde os nós do grafo representam os estados e as arestas representam as ações, cada uma possuindo, respectivamente, a recompensa pela tomada da ação e a probabilidade de se chegar ao estado seguinte.

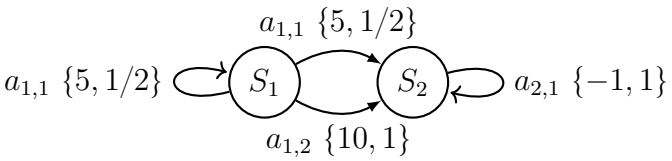


Figura 35 – Esquema de uma instância de processo de decisão de Markov. Adaptado de Puterman [14].

O treinamento foi realizado com 2 mil arquivos coletados aleatoriamente da suíte do SPEC CPU 2017 e da biblioteca `boost`, escrita em linguagem C++, utilizando uma estratégia de *Proximal Policy Optimization* (PPO) para o ajuste das políticas de tomada de decisão dos agentes. Foram treinados dois modelos distintos, um empregando apenas recompensas locais e outro utilizando recompensas globais e locais simultaneamente.

O desempenho das alocações foi medido em termos do tempo de execução do programa resultante e número de acessos à memória, comparando o resultado de 18 *benchmarks* dos SPEC CPU 2006 e 2017 utilizando-se o modelo de *reinforcement learning* e os alocadores tradicionais do LLVM (*basic*, *greedy* e *pbqp*). Os testes foram efetuados em duas máquinas: uma possuindo 32 GB de RAM, com um processador Intel Xeon SkyLake W2133 *hexa-core* (arquitetura x86), e uma de 8 GB de RAM, ARM Cortex A72 *dual-core* (arquitetura AArch64).

Em média, os resultados apresentados pelo *RL4ReAl* são comparáveis ou ligeiramente melhores do que os dos alocadores do LLVM. Na arquitetura x86, o modelo de *reinforcement learning* apresentou melhorias em tempo de execução de 1,59% e 0,96% em relação aos alocadores *basic* e *pbqp* respectivamente; em contrapartida, houve uma piora de 0,61% com relação ao alocador *greedy*. Em AArch64, o *RL4ReAl* apresentou melhorias em comparação aos três alocadores do LLVM: 1,18% sobre o *basic*, 0,22% sobre o *pbqp* e 0,26% em relação ao *greedy*. O alocador de VenkataKeerthy *et al.* também efetuou uma menor quantidade de *reloads* por *spill* em comparação aos alocadores do LLVM para ambas arquiteturas, demonstrando que os agentes de RL foram capazes de aprender uma boa política de escolha de variáveis para *spill*.

De maneira análoga, Kim *et al.* [63] propuseram a incorporação de técnicas de *reinforcement learning* na alocação de registradores via PBQP. Os autores desenvolveram um alocador para sistemas embarcados voltados para teste de chips DRAM, empregando um modelo que resolve o PBQP utilizando heurísticas baseadas em processos estocásticos e aprimoradas através do aprendizado por reforço.

No trabalho de Kim *et al.*, o grafo de resolução do PBQP é representado na forma de uma série de vetores numéricos, que são usados como entrada para uma rede convolucional em grafo (GCN). Então, o modelo realiza a alocação utilizando *Monte Carlo search tree* (MCTS), uma técnica de busca em árvore baseada em processos estocásticos, muito empregada por agentes inteligentes destinados a jogos. O algoritmo da MCTS consiste em explorar uma árvore, que corresponde às possibilidades de jogadas ou ações em um sistema, com base em conhecimento prévio e realizando simulações a cada novo nó descoberto, de modo a adaptar a base de conhecimento com os dados das melhores jogadas para se atingir um resultado pré-determinado.

O treinamento do modelo foi conduzido utilizando-se grafos aleatórios gerados pelo modelo de Erdos-Rényi. Para a avaliação do desempenho do modelo, foram empregados 24 exemplos do *suite* de testes do LLVM, e os resultados em tempo de execução foram comparados aos dos alocadores *fast*, *greedy* e *pbqp* do LLVM; a opção *fast*, que emprega alocação de registradores local, serviu como referência para os resultados obtidos. O modelo de RL de Kim *et al.* apresentou resultados piores em apenas 2 *benchmarks*, quando comparado com o alodador PBQP do LLVM.

5.4 Outros trabalhos

Alguns outros trabalhos abordam a aplicação de técnicas de *machine learning* em problemas de otimização como a coloração de grafos de maneira geral, sem necessariamente ter relação com a alocação de registradores. Schuetz *et al.* [64], por exemplo, desenvolveram um método de coloração de grafos com redes neurais inspirado por princípios da física. Goudet *et al.* [65] propuseram um modelo combinando *deep learning* e um *framework* memético, um tipo de algoritmo evolutivo, executável em GPUs.

Dodaro *et al.* [66] treinaram um agente de *deep learning* para gerar heurísticas visando resolver a coloração de grafos utilizando *answer set programming* (ASP), um paradigma de inteligência artificial para a representação de bases de conhecimento e lógica. Musliu e Schwengerer [67] identificaram 78 características dos grafos que podem indicar quais algoritmos se utilizar para realizar a coloração, e sugeriram técnicas de aprendizado de máquina para automaticamente classificar grafos e automaticamente escolher o melhor método de resolução.

No mais, a integração do aprendizado de máquina na alocação de registradores permanece ainda um horizonte relativamente inexplorado, com poucos trabalhos publicados na área. No entanto, esses seletos trabalhos já se mostram suficientemente promissores para motivar futuros esforços de pesquisa, visando desenvolver alocadores de registradores que empregam *machine learning*.

6 PRÓXIMAS ETAPAS

Com este texto finalizado, estão concluídas as fases de levantamento bibliográfico, análise das técnicas e de escrita da versão preliminar do Trabalho de Conclusão de Curso, objetivo final da disciplina de TCC I. Sendo assim, o restante deste trabalho, que engloba a realização de alguma contribuição apoiada nos conhecimentos pesquisados e aqui descritos, será desenvolvido como parte da disciplina de TCC II. A contribuição por sua vez consistirá de um trabalho de implementação, seguido de coleta de dados e análise comparativa com as ferramentas e técnicas do estado da arte. Os resultados dessa etapa serão incluídos neste texto para compor as versões para banca e final do TCC.

Dadas as tarefas definidas previamente no projeto de TCC, temos a lista atualizada de atividades, juntamente ao cronograma apresentado na Tabela 3:

1. Levantamento bibliográfico (concluído);
2. Análise das técnicas e estudo das ferramentas (concluído);
3. Escrita da versão preliminar (concluído);
4. Planejamento da implementação (em curso);
5. Implementação;
6. Coleta e análise dos resultados;
7. Escrita da versão para banca/final.

Tabela 3 – Cronograma de Execução

Ativ.\mês	ago.	set.	out.	nov.	dez.	jan.	fev.	mar.	abr.	mai.
Ativ. 1	✓									
Ativ. 2	✓	✓	✓							
Ativ. 3	✓	✓	✓	✓	✓					
Ativ. 4		•	•	•	•	•				
Ativ. 5						•	•	•		
Ativ. 6							•	•	•	
Ativ. 7						•	•	•	•	•

REFERÊNCIAS

[1] CHAITIN, G. J. Register allocation & spilling via graph coloring. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: Association for Computing Machinery, 1982. (SIGPLAN '82), p. 98–105. ISBN 0897910745.

[2] 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.

[3] WIMMER, C. Linear scan register allocation for the java hotspottm client compiler. *Master's thesis, Johannes Kepler University Linz*, 2004.

[4] 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.

[5] 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.

[6] 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.

[7] ALZUBI, J.; NAYYAR, A.; KUMAR, A. Machine Learning from Theory to Algorithms: An Overview. *Journal of Physics: Conference Series*, IOP Publishing, v. 1142, n. 1, p. 012012, nov 2018.

[8] GORBAN, A. N.; ZINOVYEV, A. Y. Principal graphs and manifolds. In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. [S.l.]: IGI Global, 2010. p. 28–59.

[9] ALPAYDIN, E. *Introduction to Machine Learning, 4th edition*. [S.l.]: MIT Press, 2020. (Adaptive Computation and Machine Learning series). ISBN 9780262043793.

[10] STEPHENSON, M. et al. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2003. (PLDI '03), p. 77–90. ISBN 1581136625.

[11] YU, Y. et al. A review of recurrent neural networks: LSTM cells and network architectures. *Neural Computation*, MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info , v. 31, n. 7, p. 1235–1270, 2019.

[12] DAS, D.; AHMAD, S. A.; KUMAR, V. Deep learning-based approximate graph-coloring algorithm for register allocation. In: IEEE. *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.], 2020. p. 23–32.

[13] VENKATAKEERTHY, S. et al. Rl4real: Reinforcement learning for register allocation. In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. New York, NY, USA: Association for Computing Machinery, 2023. (CC 2023), p. 133–144. ISBN 9798400700880.

[14] PUTERMAN, M. L. Markov decision processes. In: *Stochastic Models*. [S.l.]: Elsevier, 1990, (Handbooks in Operations Research and Management Science, v. 2). p. 331–434.

[15] MOHAMMED, M.; KHAN, M.; BASHIER, E. *Machine Learning: Algorithms and Applications*. [S.l.]: CRC Press, 2016. ISBN 9781498705387.

[16] AHO, A. et al. *Compilers: Principles, Techniques, and Tools*. [S.l.]: Addison-Wesley, 2007. (Alternative eText Formats Series). ISBN 9780321547989.

[17] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. 1st. ed. [S.l.]: Morgan Kaufmann, 1997.

[18] MITTAL, S. A survey of techniques for designing and managing cpu register file. *Concurrency and Computation Practice and Experience*, v. 29, 07 2016.

[19] BRIGGS, P. *Register allocation via graph coloring*. [S.l.]: Rice University, 1992.

[20] VERMA, M.; MARWEDEL, P. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 14, n. 8, p. 802–815, 2006.

[21] PEREIRA, F. M. Quintão; PALSBERG, J. Register allocation by puzzle solving. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 43, n. 6, p. 216–226, jun 2008. ISSN 0362-1340.

[22] CHAITIN, G. J. et al. Register allocation via coloring. *Computer Languages*, v. 6, n. 1, p. 47–57, 1981. ISSN 0096-0551.

[23] KARP, R. M. *Reducibility among Combinatorial Problems*. [S.l.]: Springer US, 1972. 85-103 p. ISBN 978-1-4684-2001-2.

[24] GAREY, M. R.; JOHNSON, D. S. The complexity of near-optimal graph coloring. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 23, n. 1, p. 43–49, jan 1976. ISSN 0004-5411.

[25] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 24, n. 7, p. 258–263, jun 1989. ISSN 0362-1340.

[26] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Rematerialization. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1992. (PLDI '92), p. 311–321. ISBN 0897914759.

[27] SHARMA, N.; SHARMA, R.; JINDAL, N. Machine learning and deep learning applications-a vision. *Global Transitions Proceedings*, v. 2, n. 1, p. 24–28, 2021. ISSN 2666-285X. 1st International Conference on Advances in Information, Computing and Trends in Data Engineering (AICDE - 2020).

[28] The LLVM Compiler Infrastructure. Acesso em: 16 de junho de 2023. Disponível em: <https://llvm.org>.

[29] ALLEN, F. E. Control flow analysis. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 5, n. 7, p. 1–19, 1970.

[30] AMD. AMD64 architecture programmer's manual volume 2: System programming. *2006*, 2006.

[31] KADY, S. E.; KHATER, M.; ALHAFNAWI, M. MIPS, ARM and SPARC-an architecture comparison. In: *Proceedings of the World Congress on Engineering*. [S.l.]: International Association of Engineers, 2014. v. 1.

[32] GEORGE, L.; APPEL, A. W. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 3, p. 300–324, may 1996. ISSN 0164-0925.

[33] BRAUN, M.; HACK, S. Register spilling and live-range splitting for SSA-form programs. In: MOOR, O. de; SCHWARTZBACH, M. I. (Ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 174–189. ISBN 978-3-642-00722-4.

[34] EISL, J. et al. Trace-based register allocation in a JIT compiler. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: Association for Computing Machinery, 2016. (PPPJ '16). ISBN 9781450341356.

[35] LAWLER, E. A note on the complexity of the chromatic number problem. *Information Processing Letters*, v. 5, n. 3, p. 66–67, 1976. ISSN 0020-0190.

[36] BJöRKLUND, A.; HUSFELDT, T.; KOIVISTO, M. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, v. 39, n. 2, p. 546–563, 2009.

[37] BRIGGS, P. et al. Coloring heuristics for register allocation. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1989. (PLDI '89), p. 275–284. ISBN 089791306X.

[38] PROTZENKO, J. *A survey of register allocation techniques*. [S.l.], 2009.

[39] JOHANSSON, E.; SAGONAS, K. Linear scan register allocation in a high-performance erlang compiler. In: SPRINGER. *International Symposium on Practical Aspects of Declarative Languages*. [S.l.], 2001. p. 101–119.

[40] TRAUB, O.; HOLLOWAY, G.; SMITH, M. D. Quality and speed in linear-scan register allocation. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1998. (PLDI '98), p. 142–151. ISBN 0897919874.

[41] 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.

[42] JR, V. H.; BICKNELL, M. Triangular numbers. *Fibonacci Quarterly*, Citeseer, v. 12, n. 3, p. 221–230, 1974.

[43] 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.

[44] ECKSTEIN, E.; KÖNIG, O.; SCHOLZ, B. Code instruction selection based on ssa-graphs. In: SPRINGER. *International Workshop on Software and Compilers for Embedded Systems*. [S.l.], 2003. p. 49–65.

[45] HAMES, L.; SCHOLZ, B. Nearly optimal register allocation with PBQP. In: SPRINGER. *Joint Modular Languages Conference*. [S.l.], 2006. p. 346–361.

[46] GOLUMBIC, M. C. *Algorithmic graph theory and perfect graphs*. [S.l.]: Elsevier, 2004.

[47] DAGAN, I.; GOLUMBIC, M. C.; PINTER, R. Y. Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, Elsevier, v. 21, n. 1, p. 35–46, 1988.

[48] WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 13, n. 2, p. 181–210, 1991.

[49] CALLAHAN, D.; KOBLENZ, B. Register allocation via hierarchical graph coloring. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 26, n. 6, p. 192–203, 1991.

[50] CHOW, F.; HENNESSY, J. Register allocation by priority-based coloring. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: Association for Computing Machinery, 1984. (SIGPLAN '84), p. 222–232. ISBN 0897911393.

[51] USAMA, M. et al. Unsupervised Machine Learning for Networking: Techniques, Applications and Research Challenges. *IEEE Access*, v. 7, p. 65579–65615, 2019.

[52] PRAKASH, V. J.; NITHYA, L. M. A survey on semi-supervised learning techniques. *CoRR*, abs/1402.4645, 2014.

[53] ENGELEN, J. E. V.; HOOS, H. H. A survey on semi-supervised learning. *Machine learning*, Springer, v. 109, n. 2, p. 373–440, 2020.

[54] KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement Learning: A Survey. *arXiv e-prints*, abr. 1996.

[55] KROGH, A. What are artificial neural networks? *Nature Biotechnology*, Nature Publishing Group US New York, v. 26, n. 2, p. 195–197, 2008.

[56] MATHEW, A.; AMUDHA, P.; SIVAKUMARI, S. Deep learning techniques: An overview. In: HASSANIEN, A. E.; BHATNAGAR, R.; DARWISH, A. (Ed.). *Advanced Machine Learning Technologies and Applications*. Singapore: Springer Singapore, 2021. p. 599–608. ISBN 978-981-15-3383-9.

[57] SHARMA, S.; SHARMA, S.; ATHAIYA, A. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology, IJEAST*, v. 4, n. 12, p. 310–316, 2020. ISSN 2455-2143.

[58] DARWIN, C. *Origin of the Species*. [S.l.]: John Murray, 1859.

[59] KOZA, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, Springer, v. 4, p. 87–112, 1994.

[60] LEMOS, H. et al. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In: IEEE. [S.l.], 2019. p. 879–885.

[61] SCARSELLI, F. et al. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, v. 20, n. 1, p. 61–80, 2009.

[62] LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis and transformation. In: IEEE. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* [S.l.], 2004. p. 75–86.

[63] 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.

[64] SCHUETZ, M. J. A. et al. Graph coloring with physics-inspired graph neural networks. *Phys. Rev. Res.*, American Physical Society, v. 4, p. 043131, Nov 2022.

[65] GOUDET, O.; GRELIER, C.; HAO, J.-K. A deep learning guided memetic framework for graph coloring problems. *Knowledge-Based Systems*, v. 258, p. 109986, 2022. ISSN 0950-7051.

[66] DODARO, C. et al. Deep learning for the generation of heuristics in answer set programming: A case study of graph coloring. In: GOTTLOB, G.; INCLEZAN, D.; MARATEA, M. (Ed.). *Logic Programming and Nonmonotonic Reasoning*. Cham: Springer International Publishing, 2022. p. 145–158. ISBN 978-3-031-15707-3.

[67] MUSLIU, N.; SCHWENGERER, M. Algorithm selection for the graph coloring problem. In: NICOSIA, G.; PARDALOS, P. (Ed.). *Learning and Intelligent Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 389–403. ISBN 978-3-642-44973-4.