Universidade Estadual de Londrina

MATHEUS PIRES VILA REAL

APLICAÇÕES DA INTELIGÊNCIA ARTIFICIAL NA GERAÇÃO DE CÓDIGO *SPILL*

LONDRINA
2024

MATHEUS PIRES VILA REAL

APLICAÇÕES DA INTELIGÊNCIA ARTIFICIAL NA GERAÇÃO DE CÓDIGO *SPILL*

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

2024

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

R288a Real, Matheus.
Aplicações da Inteligência Artificial na geração de código spill / Matheus Real. - Londrina, 2024.
109 f. : il.

Orientador: Wesley Attrot.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2024.
Inclui bibliografia.

1. Alocação de Registradores - TCC. 2. Inteligência Artificial - TCC. 3. Compiladores - TCC. 4. Geração de Código Spill - TCC. I. Attrot, Wesley. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

MATHEUS PIRES VILA REAL

APLICAÇÕES DA INTELIGÊNCIA ARTIFICIAL NA GERAÇÃO DE CÓDIGO *SPILL*

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. Fábio Sakuray
Universidade Estadual de Londrina

Prof.ª Dr.ª Helen Cristina de Mattos Senefonte
Universidade Estadual de Londrina

Londrina, 07 de maio de 2024.

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Wesley, pela mentoria intelectual e toda ajuda na realização deste trabalho. Também agradeço aos demais professores do Departamento de Computação e da Universidade por todo o conhecimento passado e aprendizado proporcionado ao longo dos últimos quatro anos, sem os quais a realização deste trabalho não teria sido possível.

Sou eternamente grato aos meus pais e familiares por todo o apoio material e emocional dado ao longo do curso, que me incentivaram e foram fundamentais para que eu tenha chegado até este momento de conclusão. Por fim, agradeço aos meus colegas do curso pelas boas memórias e pela companhia em todos os momentos de minha trajetória acadêmica.

"Feliz é o homem que persevera na provação, porque depois de aprovado receberá a coroa da vida, que Deus prometeu aos que o amam."
(Bíblia Sagrada, Tiago 1:12)

REAL, M. **Aplicações da Inteligência Artificial na geração de código *spill***. 2024. 109f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

RESUMO

A alocação de registradores é uma das otimizações de código mais significativas do processo de compilação de um programa, sendo responsável por mapear as variáveis definidas no código aos registradores da arquitetura-alvo ou a posições na memória principal. Uma abordagem eficiente minimiza a utilização da memória, de modo a promover tempo de execução e gasto energético menores. No entanto, os algoritmos tradicionais para alocação de registradores não são exatos e somente obtêm soluções aproximadas, dependendo do uso de heurísticas para que produzam código de qualidade. Com o crescimento recente da área de inteligência artificial (IA), surge a perspectiva de utilização das técnicas de IA para o aprimoramento e o desenvolvimento de novas heurísticas de alocação. Nesse sentido, este trabalho realizou uma investigação do estado da arte sobre alocação de registradores, inteligência artificial e das possíveis aplicações combinando ambas as áreas. Então, foi proposta a implementação de um modelo de programação genética, técnica de IA, para a obtenção automática de heurísticas. Os experimentos resultaram em melhorias no tempo de execução de 2,45%, 1,29% e 0,89%, para um conjunto de *benchmarks* selecionados, demonstrando a aplicabilidade da inteligência artificial na alocação de registradores.

Palavras-chave: compiladores, inteligência artificial, código *spill*, alocação de registradores, otimização, algoritmos evolutivos, programação genética

REAL, M. **Artificial Intelligence applications in spill code generation**. 2024. 109p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

ABSTRACT

Register allocation is one of the most significant code optimizations in a program's compilation process, where variables defined in the code are mapped to physical registers of the target architecture or positions in memory. An efficient approach minimizes memory usage in order to promote lower runtime and energy expenditure. However, the traditional register allocation algorithms are not exact and provide only approximate solutions, relying on heuristics to produce quality code. With the recent growth of Artificial Intelligence (AI), the prospect of using AI techniques to enhance and develop new allocation heuristics emerges. In that regard, this work conducted an investigation about the state of the art in register allocation, Artificial Intelligence, and potential applications combining both areas. Subsequently, it was proposed the implementation of a model using genetic programming, an AI technique, aimed to accomplish automatic derivation of heuristics. The experiments resulted in runtime improvements of 2.45%, 1.29%, and 0.89% for a selected set of benchmarks, demonstrating the applicability of Artificial Intelligence in register allocation.

Keywords: compilers, artificial intelligence, spill code, register allocation, optimization, evolutionary algorithms, genetic programming.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

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

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 [20].

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 [21].

Dentre as etapas de otimização, destaca-se a alocação de registradores, onde o compilador deve encontrar estruturas de memória para armazenar os valores utilizados ao longo do programa. Os registradores são componentes microarquiteturais diretamente disponíveis ao processador, e caracterizam-se como as estruturas de memória de mais rápido acesso [22]. De preferência, o compilador deve mapear as variáveis do código para registradores, otimizando a performance do programa. Contudo, devido à quantidade limitada de registradores disponíveis, surge a possibilidade de não haverem registradores o suficiente para comportar todos os valores e resultados intermediários em uso ao mesmo tempo [20].

Nesses casos, há a necessidade de se armazenar algumas variáveis na 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* [23]. 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 [24], e um alocador sofisticado pode proporcionar uma melhora de até 250% no tempo de execução de um programa comparado a um alocador simples [25].

Os algoritmos mais bem consolidados consistem nos de alocação via coloração de grafos e via *linear scan* [26]. Na abordagem via coloração de grafos, os valores em uso no programa, abstraídos na forma de registradores virtuais, são representados na como os vértices de um grafo não-direcionado a ser colorido; uma aresta no grafo indica a impossibilidade dos dois vértices conectados ocuparem o mesmo registrador, e as cores

representam os registradores físicos disponíveis. Na abordagem via *linear scan*, a alocação dos registradores físicos é realizada através de um único percurso pelo conjunto de tempos de vida correspondentes aos registradores virtuais, em uma ordem linear pré-determinada [2].

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 [27], e descobrir a mera quantidade de cores necessárias para colorir um grafo é um problema NP-*hard* [28]. O *linear scan*, por sua vez, é uma solução gulosa para o problema da alocação, e depende fortemente da ordem empregada no percurso e das heurísticas utilizadas [2, 3].

Ademais, uma implementação ingênua pode introduzir *spill code* desnecessário ao longo de todo o tempo de vida de uma variável [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 à memória [29]. Por isso, ao longo dos anos, soluções para se contornar as dificuldades de se gerar código *spill* de maneira eficiente foram propostas.

Algumas técnicas propõe a realização de ajustes nos algoritmos originais, visando tornar a introdução de instruções `load/store` mais precisa nos pontos de interferência necessários [1, 23, 30, 5, 6, 31]. Outras abordagens introduziram a utilização de heurísticas para calcular estimativas de custo de *spill*, visando melhorar a qualidade das escolhas de quais variáveis mapear para a memória principal [1, 29, 32]. Infelizmente, o desenvolvimento e aprimoramento de heurísticas de alocação ainda é feito de maneira manual, muitas vezes através de um processo tedioso de tentativa e erro [12].

A inteligência artificial (IA), por sua vez, é a área da ciência da computação encarregada do desenvolvimento de modelos e técnicas para realizar tarefas que, tipicamente, requereriam inteligência humana. Isso é atingido apoiando-se em técnicas de busca e otimização, matemática, lógica formal, estatística, e até mesmo economia e biologia. Esses fundamentos tornam as abordagens de IA muito úteis para solucionar problemas para os quais uma solução exata não está disponível [33], como a alocação de registradores, através de métodos como aprendizado de máquina [9, 34] e algoritmos evolutivos [35, 18, 11].

Nas últimas décadas, a inteligência artificial como um todo vem avançando consideravelmente, graças ao crescimento do poder computacional 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 IA. 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 [12], Das, Ahmad e Kumar [14], e VenkataKeerthy *et al.* [15].

Este trabalho se propõe a investigar a aplicação das técnicas de inteligência artificial 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 em uma revisão bibliográfica sobre a área, serão propostos métodos para a obtenção heurísticas que utilizam IA. É esperado que o modelo desenvolvido apresente melhorias no código produzido ao final da etapa de alocação de registradores, apresentando um desempenho ao menos comparável aos presentes em compiladores tradicionais, como o LLVM [36].

O Capítulo 2 apresenta em detalhes os principais algoritmos disponíveis para resolução da alocação de registradores, enquanto o Capítulo 3 expõe as técnicas de minimização de *spill code* e as heurísticas desenvolvidas ao longo dos anos para otimizar a performance dos alocadores. No Capítulo 4, é feita uma revisão das técnicas de inteligência artificial consideradas para este trabalho e utilizadas nos trabalhos correlatos analisados no Capítulo 5, que expõe o estado da arte da aplicação de IA no problema de alocação de registradores. Os Capítulos 6 e 7 apresentam, respectivamente, a contribuição proposta neste trabalho e os resultados dos experimentos realizados. Por fim, no Capítulo 8 são feitas as considerações finais e são apontados os prospectos para a realização de trabalhos futuros.

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 [20].

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* [37].

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 [21].

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

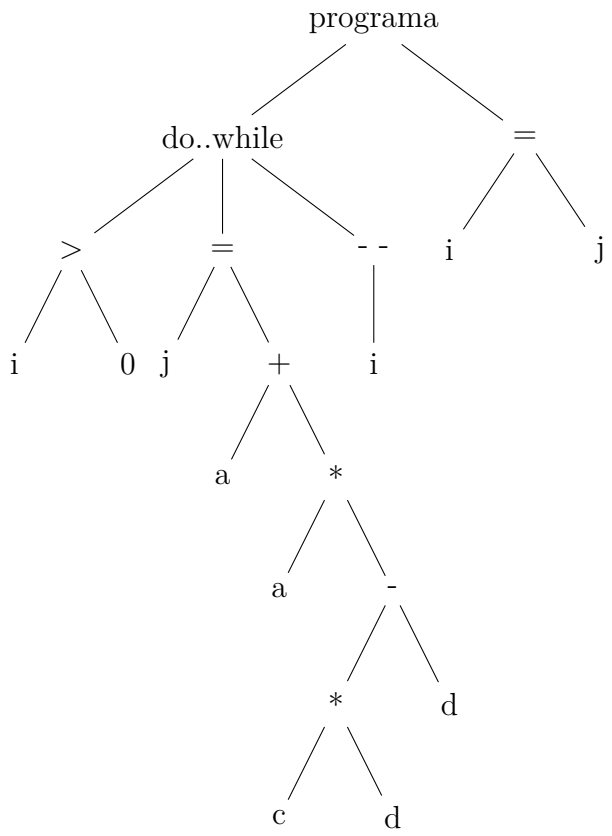


Figura 2 – Árvore sintática abstrata representando o código da Figura 1.

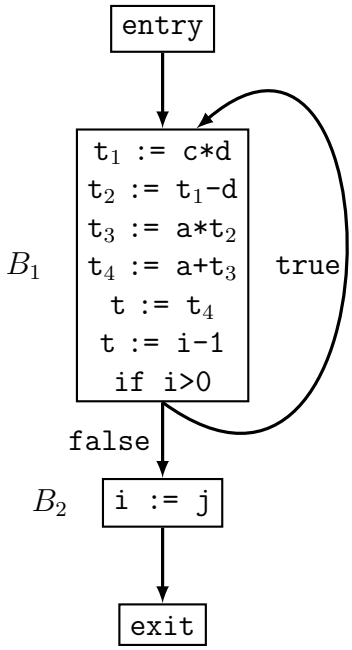


Figura 3 – Grafo de controle de fluxo equivalente gerado a partir da árvore da Figura 2, com os registradores virtuais `i` e `j`, correspondentes às variáveis, além de $\texttt{t}_1$, $\texttt{t}_2$, $\texttt{t}_3$ e $\texttt{t}_4$ que representam os valores temporários da expressão aritmética calculada dentro do laço de repetição.

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 [21].

2.1 *Liveness Analysis* e Interferências

Em posse de uma representação intermediária contendo as variáveis, também denominadas 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 [38]; arquiteturas RISC como ARMv8, MIPS e SPARC apresentam respectivamente 31, 30 e 32 registradores [39].

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.

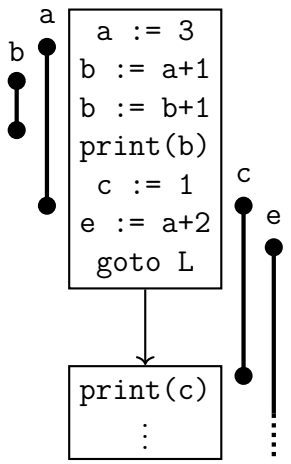


Figura 4 – Dois blocos básicos contendo os registradores virtuais, ou variáveis, `a`, `b`, `c` e `e`, com seus respectivos *live ranges* indicados.

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 [20].

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 vivos em registradores ao mesmo tempo, 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 [40, 1, 23].

2.2 Geração de Código *Spill*

Quando o número de *live ranges* vivos em um mesmo ponto do programa ultrapassa a quantidade de registradores físicos disponíveis, faz-se necessário armazenar algum deles na memória principal. Para isso, são introduzidas as instruções de acesso a memória `store`, para escrever, e `load` para recuperar um valor em memória. O número de registradores físicos necessários para comportar os *live ranges* em um ponto do código é uma métrica conhecida como pressão de registradores [41].

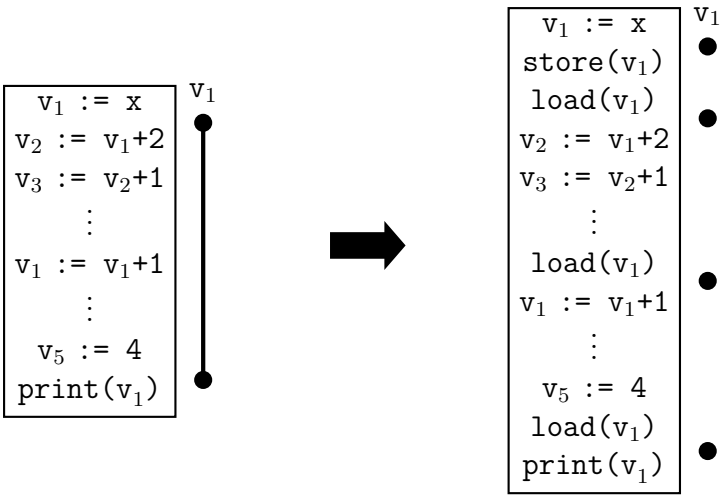


Figura 5 – Exemplo de bloco básico antes e depois do *spill* do *live range* $\mathtt{v}_1$, mostrando a fragmentação de seu tempo de vida pelas instruções de acesso à memória.

Essas instruções são denominadas código *spill*, e tem por efeito fragmentar o tempo de vida *live range* escolhido ao redor do ponto de grande pressão, a fim de reduzi-la e possibilitar o armazenamento de todos os valores usados no programa [42]. Ao longo do curso de execução de um programa, uma mesma variável pode ser ser armazenada em registra-

dores e posteriormente ser enviada para a memória, e ter seu *live range* particionado [43]. 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. A tarefa de 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 de quais variáveis selecionar para *spill* e de posicionamento 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 [42], 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 [43].

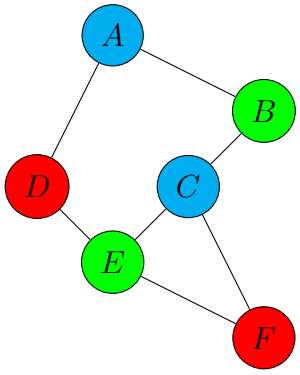


Figura 6 – Exemplo de grafo de interferência colorido com três cores.

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 $e = (v_i, v_j)$ 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 [27] — determinar a k-colorabilidade de um grafo

possui complexidade exponencial e logo torna-se inviável para casos que envolvem muitos registradores virtuais [44, 45], e a mera tarefa de determinar o número mínimo de cores para se colorir um grafo é um problema NP-*hard* [28]. 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*;

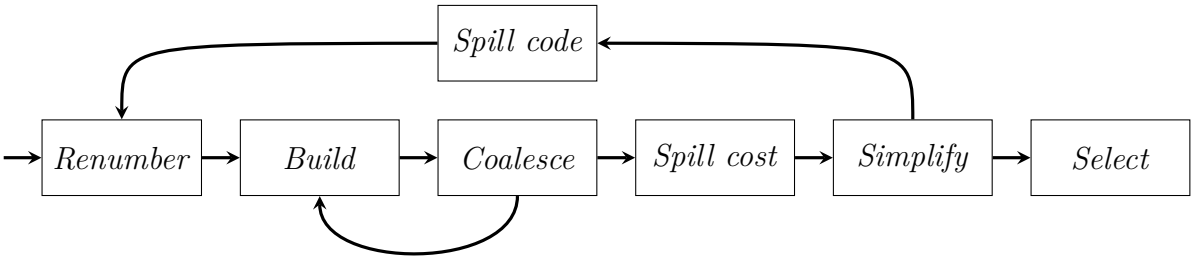


Figura 7 – Esquema do algoritmo de Chaitin [1].

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 [23] 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 [46], 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.

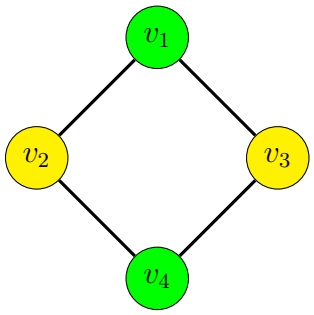


Figura 8 – Grafo 2-colorível que seria incorretamente colorido pelo alocador de Chaitin.

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.* [30] 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.

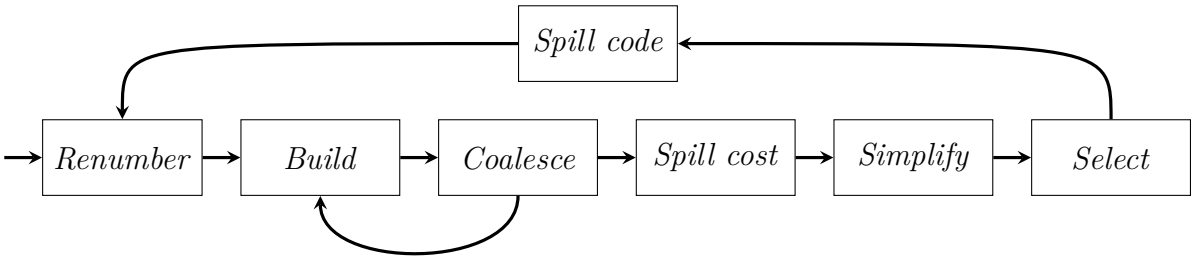


Figura 9 – Esquema do algoritmo de Briggs, com a etapa de geração de *spill code* adiada.

2.3.3 *Iterated Register Coalescing*

Em 1996, George e Appel [40] 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 [26]. A Figura 10 apresenta um esquema do funcionamento do alocador de George e Appel:

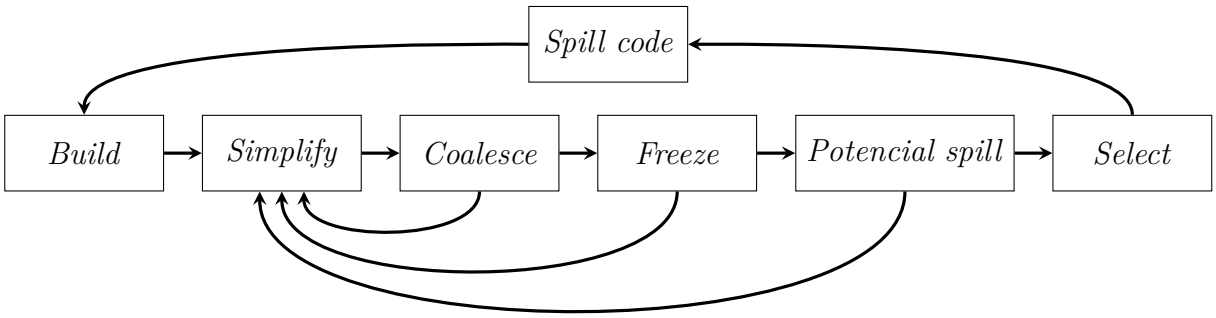


Figura 10 – Esquema do Iterated Register Coalescing, apresentando a etapa de *freeze*.

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 menor. Isso ocorre pois o algoritmo de *linear scan* apresenta uma complexidade linear, em contrapartida à abordagem por coloração de grafos que tem complexidade quadrática [47]. 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 de intervalos vivos, ou *live intervals*, mostrados na Figura 11. 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].

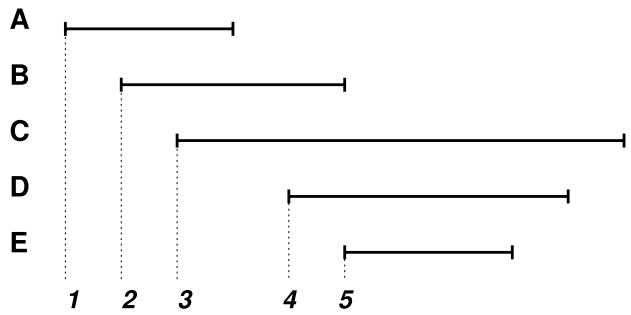


Figura 11 – Exemplo de um conjunto de cinco *live intervals*. Extraído de Poletto e Sarkar [2].

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.

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 [43].

Em 1998, Traub *et al.* [31] 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 Algoritmo do *linear scan*. Adaptado de Poletto e Sarkar [2].

Dados: A: lista de ativos, R: conjunto de registradores físicos, I: conjunto de *live intervals*

```
procedimento LinearScan
    A ← {}, ordenado em ordem crescente de pontos finais
    livres ← R
    para todo live interval i ∈ I faça
        ExpirarIntervalos(i, A)
        A ← A ∪ {i}
        se |A| < |R| então
            registrador[i] ← um registrador r ∈ livres
            livres ← livres − {r}
        senão
            Spill(i, A)
função ExpirarIntervalos(i, A)
    para todo live interval a ∈ A faça
        se fim[a] ⩾ inicio[i] então
            retorne
        A ← A − {a}
        livres ← livres ∪ {registrador[a]}
função Spill(i, A)
    spill ← último live interval em A
    se fim[spill] > fim[i] então
        registrador[i] ← registrador[spill]
        spill é enviado para a memória principal
        A ← A − {spill}
    senão
        spill é enviado para a memória principal
```

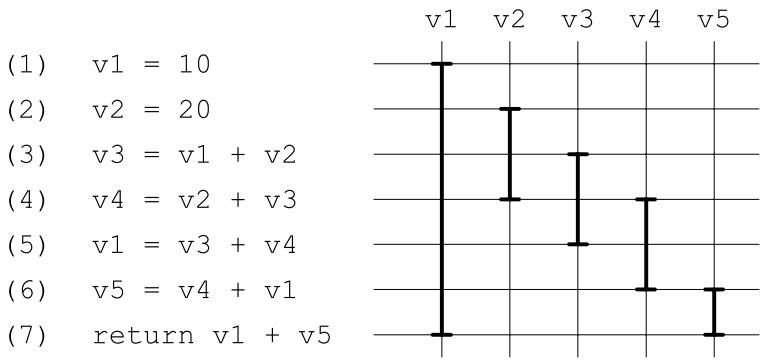


Figura 12 – Exemplo em pseudocódigo mostrando os *live intervals* no *linear scan* tradicional. Figura extraída de Wimmer *et al.* [3].

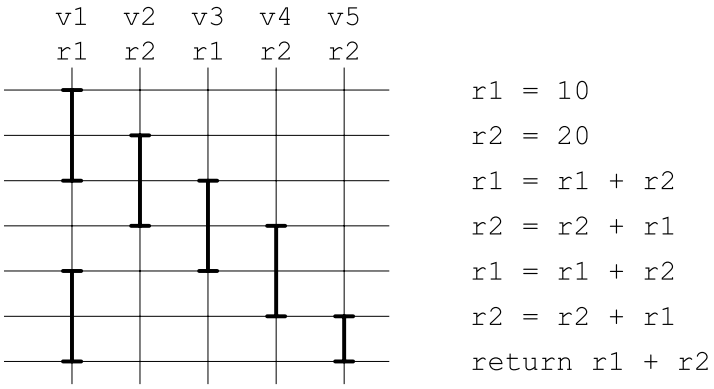


Figura 13 – O mesmo exemplo da Figura 12 com os *live intervals* do *second-chance bin-packing*, e a alocação de registradores final. Extraído de Wimmer [3].

2.5 Alocação via PBQP

Em 2002, Scholz e Eckstein [48] 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 mostra como $\vec{c_v}$ é computado:

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

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 do vetor $\vec{c_v}$ 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 $\vec{x_u} C_{uv} \vec{x_v}^T$, mostrada na 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 da Fórmula 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 $\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n = \frac{(n^2+n)}{2}$ [49].

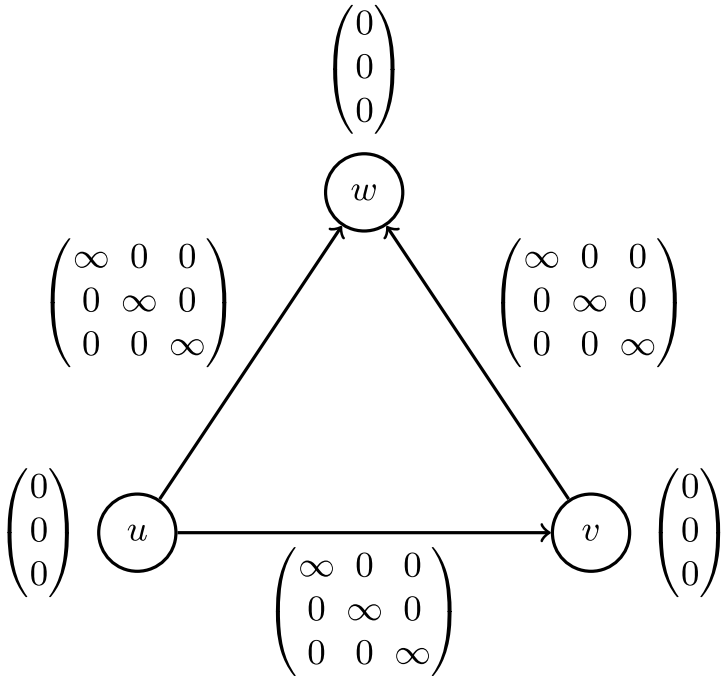


Figura 14 – Grafo de resolução do PBQP considerando 3 registradores virtuais. Extraído de Buchwald *et al* [4]

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 [50, 51]:

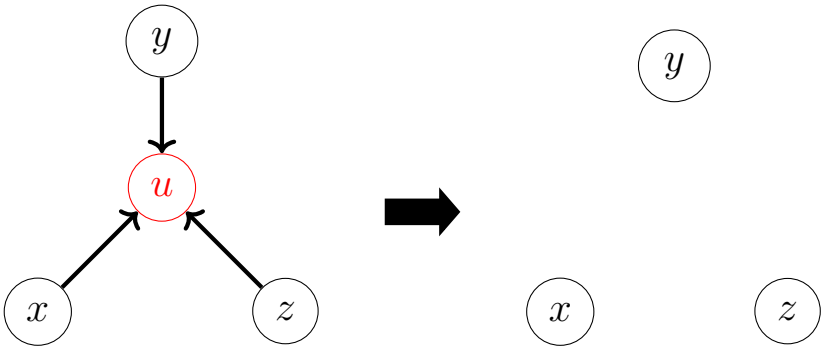
1. ***RE*** — remoção de arestas independentes, possuindo uma matriz de custos que pode ser decomposta em dois vetores $\vec{u}$ e $\vec{v}$, ou seja, cada entrada da matriz c_{ij} tem custos $u_i + v_j$. Essas arestas podem ser removidas ao se somar $\vec{u}$ e $\vec{v}$ aos vetores de custo dos vértices 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 [52] 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 [24], e a alocação de registradores pode impactar o tempo de execução de um programa em até 250% [25].

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, 29].

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 [12].

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 [42].

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 [29].

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

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.* [29] 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:

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

onde $largura(i)$ é o número de variáveis vivas no momento de execução da instrução i. De maneira intuitiva, $área(v)$ 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.* [29] 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 [53, 54]. 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.* [29] 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.* [29]

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 [42], Bernstein [29] e Briggs [30], 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].

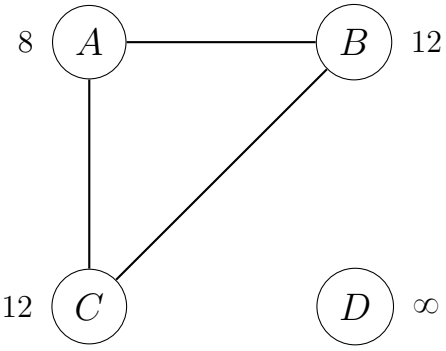


Figura 16 – Grafo de interferência correspondente ao código da Figura 15, com os custos de *spill* indicados.

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.

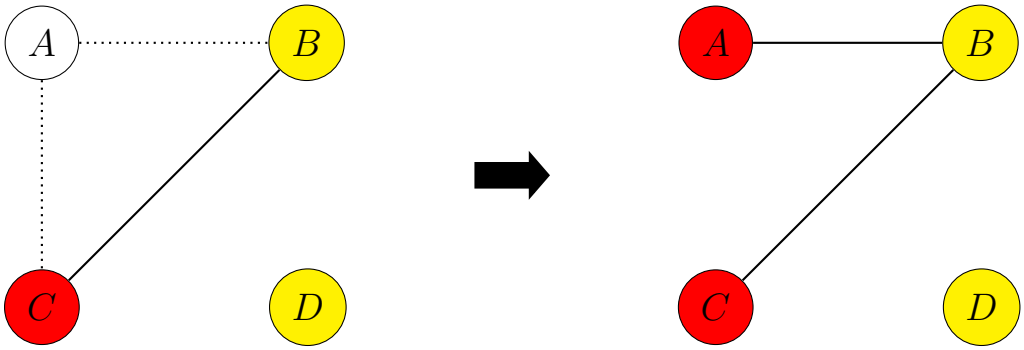


Figura 17 – Coloração do exemplo da Figura 16, considerando a estratégia de *spilling* por região de interferência. Adaptado de Bergner *et al.* [5].

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

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

}

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].

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*.

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.* [30] 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.* [30] 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 $\texttt{y}_3$ que pode receber tanto o valor de $\texttt{y}_1$ quanto $\texttt{y}_2$, a depender de qual bloco condicional será acessado em tempo de execução.

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

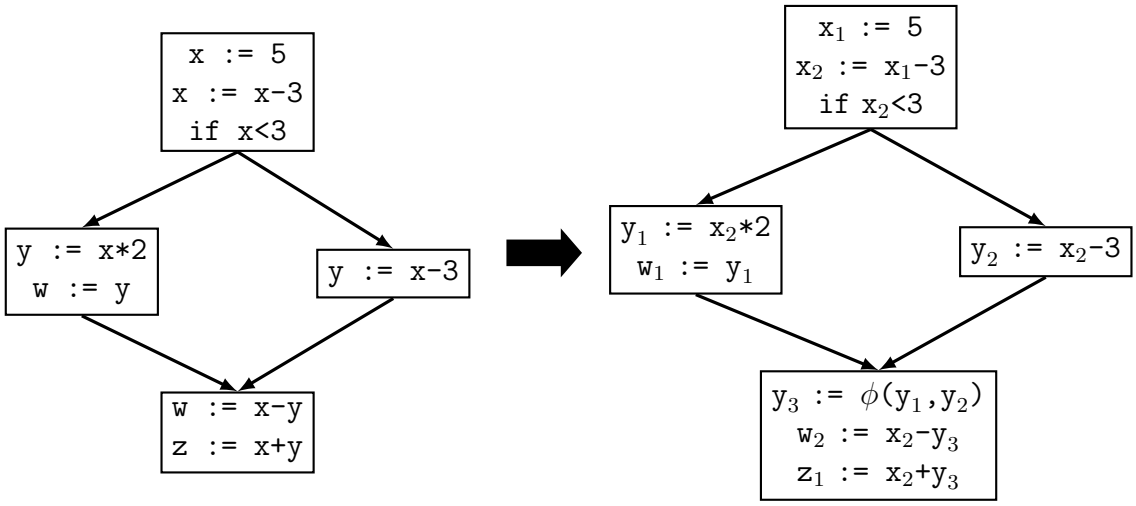


Figura 19 – Exemplo de código na forma tradicional e após a conversão para forma SSA.

ao *constant propagation* de Wegman e Zadeck [55], 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.

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 [23], como coalescimento e coloração com a mesma cor de variáveis unidas por cópia.

O algoritmo de Briggs *et al.* foi testado em 70 benchmarks e apresentou 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 [30].

3.5 Coloração de Grafos Hierárquica

Em 1991, Callahan e Koblenz [56] descreveram uma técnica de alocação de registradores que emprega uma 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 $G = (B, E, \mathit{start}, \mathit{stop})$, onde B é o conjunto dos blocos básicos, E é o conjunto de arestas entre os blocos e $\mathit{start}, \mathit{stop} \in B$ representam respectivamente os pontos de início e fim do fluxo de execução, é definida uma árvore hierárquica T.

Cada nó $t \in T$ é um membro da família de subconjuntos de B tal que $\mathit{blocos}(t) = \{b_0, b_1, \ldots, b_n\}$ e $\mathit{arestas}(t) = \{e_0, e_1, \ldots, e_n\}$, 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 $\mathit{blocos}(t)$. 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 $\mathit{blocos}(t_0) = \{\mathit{start}, \mathit{stop}\}$. 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.

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 $\mathit{blocos}(t)$, que são denominadas variáveis locais, e a coloração é feita de maneira convencional seguindo um algoritmo ao estilo de Briggs [46]. Para todos os fins, as variáveis globais a t — isto é, que são definidas em

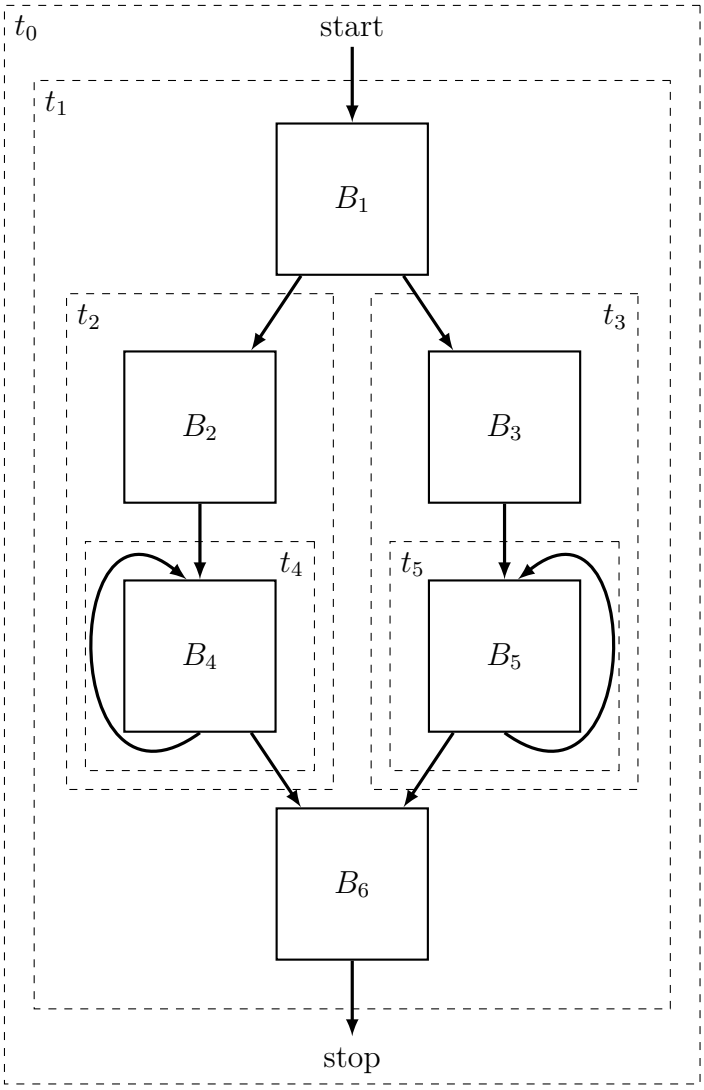


Figura 20 – Exemplo de CFG particionado em porções correspondentes aos nós $t_0, t_1, \ldots, t_5$ da árvore hierárquica.

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:

1. Duas variáveis locais ao nó interferem em algum ponto de seus blocos;

2. Uma variável global interfere com uma variável de resumo ou outra variável global;

3. Uma variável está viva em uma subárvore de t mas não consta nas variáveis de resumo vindas da subárvore;

4. 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 realizasse a compilação. Eles observaram o aproveitamento de tais recursos ao realizar experimentos com um compilador de Fortran [56], que confirmaram 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 [23, 30, 32]. 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.

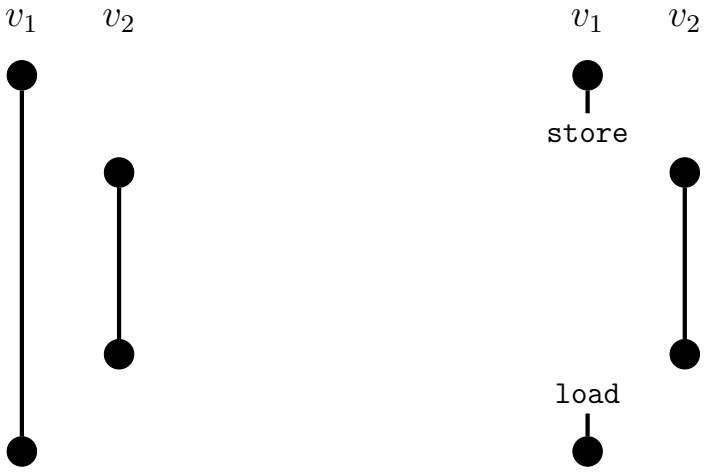


Figura 21 – Exemplo de *splitting*. Nesse caso, v_1 é dividido ao redor de v_2, eliminando a interferência.

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.

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.

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.

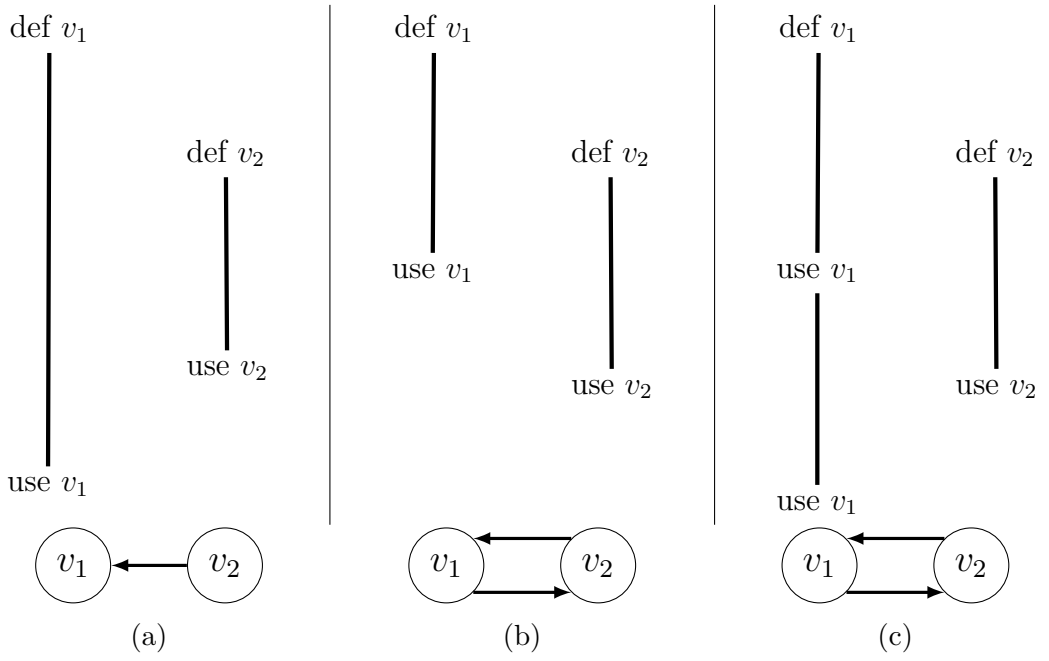


Figura 22 – Relações de contenção entre dois *live ranges* e seus respectivos grafos. Adaptado de Cooper *et al.* [6]

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

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].

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 INTELIGÊNCIA ARTIFICIAL

A inteligência artificial (IA) é uma área da ciência da computação que se encarrega de pesquisar e desenvolver sistemas computacionais capazes de desempenhar tarefas que, tipicamente, requereriam a inteligência humana. Os sistemas de IA são projetados para imitar ou mesmo superar os humanos em determinados domínios, exibindo habilidades análogas de raciocínio, representação de conhecimento, planejamento, entendimento de linguagem natural, percepção, aprendizagem e a capacidade de se mover e manipular objetos [33].

Para atingir esses objetivos, os pesquisadores e desenvolvedores de IA apropriaram-se de uma vasta gama de técnicas, incluindo busca e otimização matemática, lógica formal, redes neurais artificiais e métodos baseados em estatística, pesquisa operacional e economia. Graças a esses fundamentos, as abordagens de IA são particularmente úteis para problemas nos quais uma solução algorítmica exata não está disponível, incluindo problemas de otimização NP-completos [33] como a alocação de registradores.

Sendo assim, este capítulo tem por objetivo fundamentar os conceitos por trás das principais técnicas de inteligência artificial empregadas nas pesquisas envolvendo alocação de registradores, que serão abordadas no Capítulo 5. Em específico, serão explanadas as áreas de aprendizado de máquina e algoritmos evolutivos.

4.1 Aprendizado de Máquina

O aprendizado de máquina ou *"machine learning"* (ML), em inglês, é uma área da inteligência artificial que se encarrega do desenvolvimento de agentes inteligentes que se tornam aptos a realizar tarefas aprimorando-se a partir de dados, ao invés de serem explicitamente programados. Esse processo de aprimoramento é denominado "aprendizado" ou "treinamento", e envolve a utilização de um conjunto de dados a respeito da tarefa a ser efetuada e que são utilizados para melhorar a precisão do agente [7].

O aprendizado de máquina é utilizado em situações em que há uma grande disponibilidade de dados. Através da análise desses dados e apoiando-se em fundamentos da estatística e probabilidade, é possível obter uma aproximação suficientemente precisa dos resultados esperados [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.

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

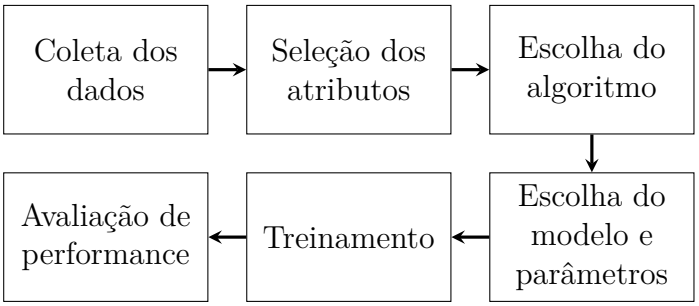


Figura 23 – Esquema ilustrando o desenvolvimento de um modelo genérico de *machine learning*. Adaptado de Alzubi *et al.* [7].

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.

O processo de aprendizado em si varia conforme as informações disponíveis e as demandas da tarefa, e consiste num processo iterativo e contínuo. Ele pode ou não envolver o tratamento de grandes quantidades de dados por parte de um intérprete humano, ou mesmo por outro modelo de inteligência artificial, e depende das decisões de projeto tomadas pelos desenvolvedores de "machine learning" [9, 34]. Os principais paradigmas de aprendizado serão explorados nas subseções seguintes.

4.1.1 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 [17, 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 [17].

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.

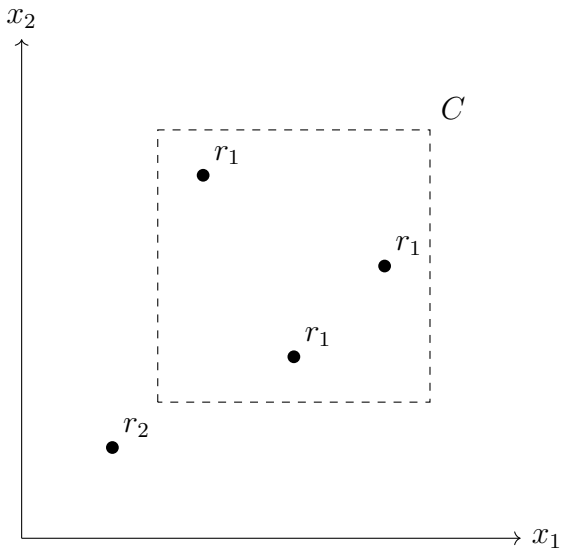


Figura 24 – Análise gráfica considerando um conjunto de treinamento com 4 elementos, formados pelos parâmetros de entrada $\vec{X} = [x_1, x_2]^T$ e rótulos $\{r_1, r_2\}$.

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.* [17]

4.1.2 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 [17].

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 [57].

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, 17], 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.

4.1.3 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 [17].

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

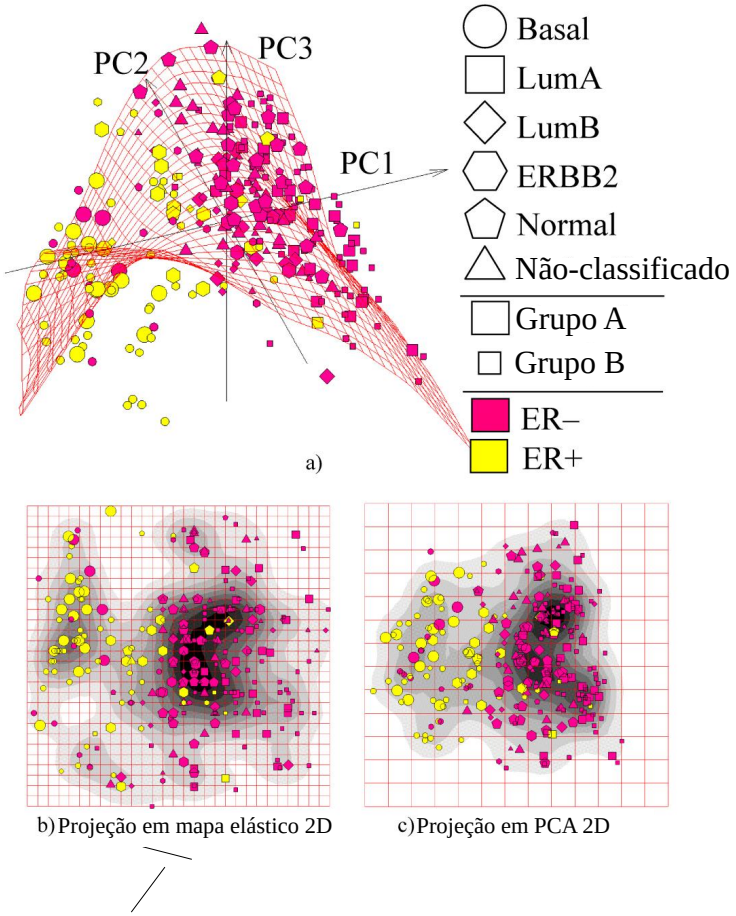


Figura 25 – *Plot* de um conjunto de dados de *microarray* de câncer de mama usando mapas elásticos, empregando técnicas de análise de componentes principais. Extraído de Gorban e Zinovyev [8].

Alguns dos principais métodos de treinamento empregados no aprendizado semi-supervisionado incluem, dentre outros [58, 59]:

- **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;

- ***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;

- ***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.

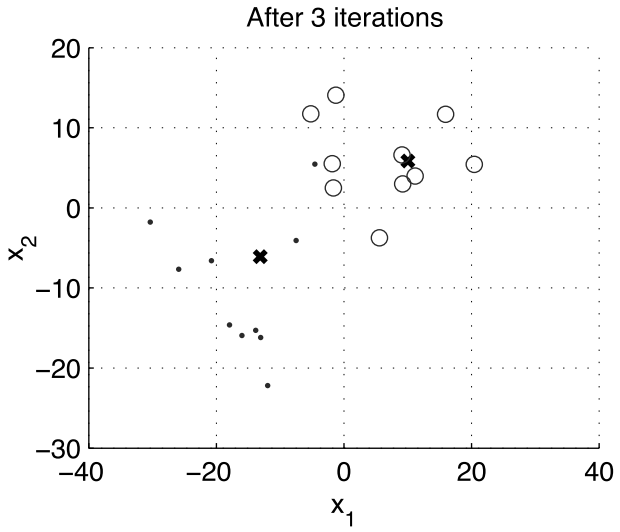


Figura 26 – Exemplo de aplicação do *k-means*. Os centroides dos *clusters* são indicados com um "×" no gráfico. Extraído de Alpaydin [9].

4.1.4 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 do sistema. A política $\pi(a, s)$ retorna a probabilidade da ação $a \in A$ ser tomada dado o estado $s \in S$ do ambiente [60].

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 [17, 60], sendo que esse processo é ilustrado pela Figura 27.

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

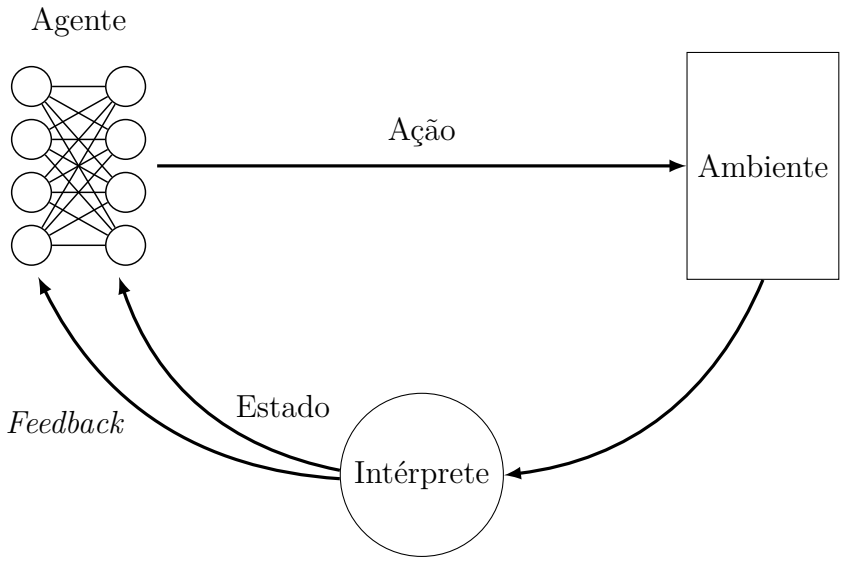


Figura 27 – Esquema do processo de treinamento via aprendizado por reforço.

do sistema muitas vezes ocorre simultaneamente ao aprendizado [60].

4.2 Redes Neurais Artificiais e Aprendizado Profundo

As redes neurais artificiais (ANNs) constituem uma classe de modelos de inteligência artificial que têm como inspiração para sua estrutura o funcionamento do cérebro humano. As redes neurais são compostas de uma série de unidades chamadas neurônios, que individualmente realizam operações matemáticas simples, e de sinapses que representam as conexões entre os neurônios.

Cada neurônio artificial funciona realizando uma soma ponderada dos valores de entrada associados cada um ao peso da sinapse, e computando uma função matemática, chamada de função de ativação, que produz uma saída propagada para a camada seguinte. Sendo assim, uma rede neural possui ao menos uma camada de entrada e uma camada de saída, que apresenta o resultado [61, 62]. A Figura 28 esquematiza um neurônio artificial que recebe como entrada os valores x_1, x_2, ..., x_n, cada qual associado a um peso w_1, w_2, ..., w_n. O somatório é adicionado a um valor b conhecido como viés ou limiar, e é submetido a uma função de ativação não-linear f.

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 nos quais 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

[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.

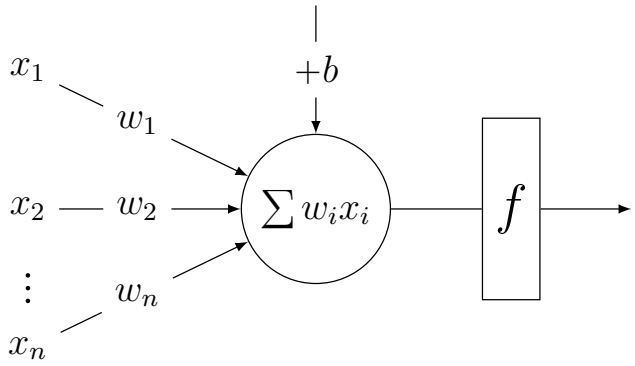


Figura 28 – Esquema do funcionamento básico de um neurônio artificial.

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 [61, 9].

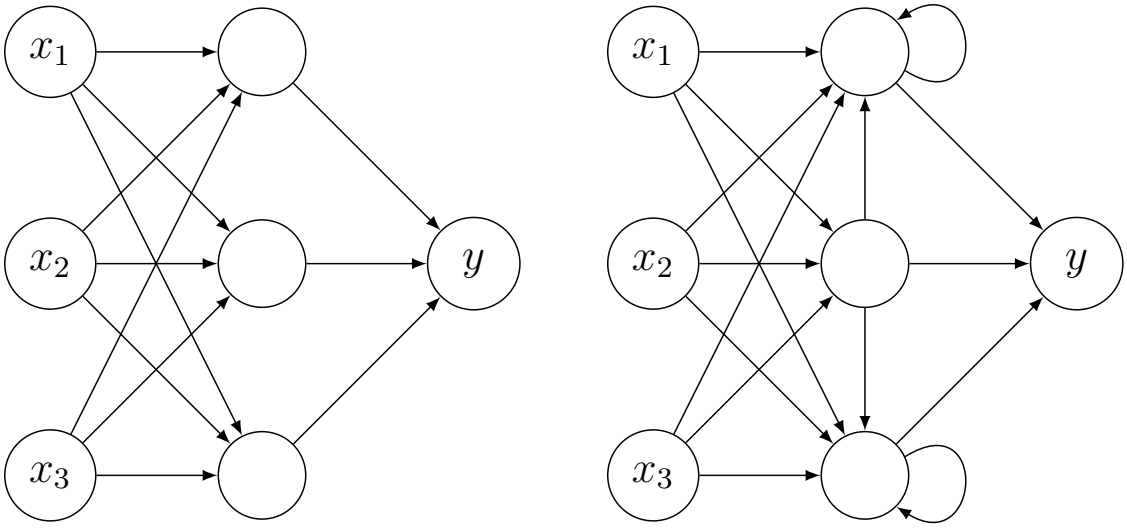


Figura 29 – Uma rede neural *feed-forward* e uma rede neural recorrente, na esquerda e direita respectivamente.

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 29, 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 [62].

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 [63]:

- ***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 produzem um resultado muito próximo do esperado [61].

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 [34].

4.3 Algoritmos Evolutivos

Os algoritmos evolutivos constituem uma família de algoritmos para otimização meta-heurística que se inspiram no conceito de evolução biológica para encontrar soluções. Eles são utilizados na área da inteligência artificial para aperfeiçoar heurísticas de problemas de otimização não-linear complexos, com um funcionamento semelhante ao dos algoritmos de busca. Contudo, os algoritmos evolutivos trazem como diferencial a capacidade de buscar soluções em um espaço de amostragem aleatória, se caracterizando como um método de *Monte Carlo*[1] [18].

Servindo como base para a computação evolutiva, a evolução biológica é um fenômeno constatado por vários cientistas independentemente ao longo de séculos, mas formalizada mais notavelmente por Charles Darwin [64]. Funcionando como um método de otimização, o processo evolutivo produziu toda a biodiversidade existente no planeta com suas sofisticadas, mas não perfeitas, soluções para a sobrevivência e reprodução dos seres vivos [35]. Contudo, do ponto de vista biológico, a evolução pode ser definida simplesmente como a mudança das frequências de alelos em uma população ao longo do tempo [18].

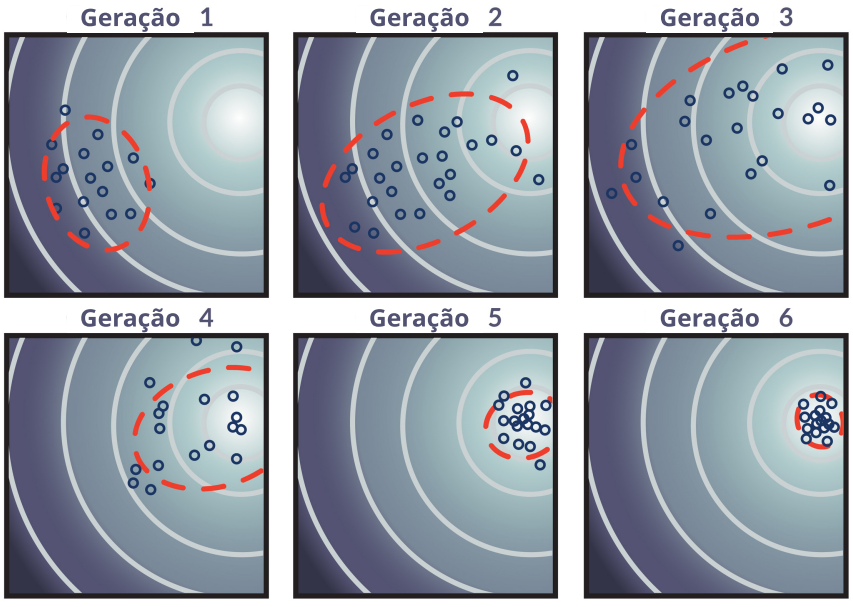


Figura 30 – Paisagem de *fitness* projetada em duas dimensões. A região mais clara indica um ótimo local, e os pontos demarcam os indivíduos que compõe a população. Ao longo de seis gerações, é possível visualizar a convergência em direção ao ponto de maior *fitness*. Extraído de um trabalho de Corpus *et al.* que utilizou estratégias evolutivas (ES) [10].

A mudança na configuração genética de uma população ocorre através de dois principais mecanismos: variação e seleção. A variação é o processo pelo qual novos alelos

[1] Os métodos de Monte Carlo são técnicas estocásticas que buscam estimar ou simular resultados determinísticos por meio de sucessivas amostragens aleatórias [33].

são inseridos na população. Isso é possível graças à reprodução sexuada e de mutações pontuais, que modificam os genomas dos indivíduos de modo a introduzir variabilidade genética. Já o processo de seleção atua reduzindo a variabilidade genética, literalmente selecionando quais alelos se perpetuam na população com base na aptidão reprodutiva dos indivíduos que os possuem [18].

Os algoritmos evolutivos, de maneira análoga, operam em populações de estruturas de dados. A variação insere alterações aleatórias em uma população de estruturas de dados recombinando partes de diferentes estruturas e ocasionalmente criando pequenas alterações pontuais. Esses dois processos são chamados respectivamente de cruzamento e mutação, e juntos são referidos como operadores de variação. A seleção é realizada com qualquer algoritmo que favoreça estruturas de dados com base em um critério de aptidão, ou *fitness* [18, 11].

Algoritmo 3 Algoritmo evolutivo genérico. Adaptado de Ashlock [18].

```
procedimento AlgoritmoEvolutivo
    P ← inicialize uma população aleatória de estruturas de dados
    repita
        para todo indivíduo i ∈ P faça
            fitness[i] ← AvaliarFitness(i)
        P′ ← Seleção(P, fitness)
        P ← Variação(P′)
    até o limite de gerações ser atingido
```

O Algoritmo 3 exemplifica o funcionamento básico de um algoritmo evolutivo genérico. Um população de indivíduos é inicializada com uma amostragem aleatória de estrutura de dados, chamados de genomas ou cromossomos. Então, por um número determinado de gerações, são avaliados os valores de *fitness* da população; o operador de seleção elimina os indivíduos com os piores *fitness* e a variação é aplicada sobre os indivíduos resultantes de modo a produzir novos indivíduos. Cada categoria de algoritmo evolutivo é caracterizada por uma combinação de representação e operadores, que serão abordados posteriormente nesta seção [18, 11].

Com o passar das gerações, dado que boas técnicas de exploração e seleção de indivíduos aptos foram empregadas, haverá uma convergência da população em direção aos ótimos locais da paisagem adaptativa[1] do problema, como mostrado pela Figura 30. Existem várias implementações diferentes de operadores de cruzamento, mutação e seleção, além das múltiplas possibilidades de escolha da estrutura de dados que representa os indivíduos [35, 11].

[1] Conceito emprestado da biologia evolutiva, consiste na representação multidimensional da aptidão reprodutiva, ou *fitness*, em função das variáveis sob processo evolutivo [11].

4.3.1 Cruzamento

O operador de cruzamento, ou *crossover*, é o principal responsável por introduzir variabilidade genética na população, ao permitir que os indivíduos troquem material genético para produzir novos genomas. Seja G um cromossomo contido na população de estruturas, o operador de cruzamento χ é definido como o mapeamento de dois elementos no domínio, denominados ancestrais, para um (Definição 4.1) ou dois (Definição 4.2) pontos na imagem denominados descendentes [18].

$$\chi : G \times G \to G \tag{4.1}$$

$$\chi : G \times G \to G \times G \tag{4.2}$$

A forma geral de um cruzamento consiste em particionar os genomas de ambos os ancestrais ao redor de um ou mais pontos escolhidos aleatoriamente, denominados lócus (loci no plural), e recombinar as partições para gerar novos genomas. Contudo, implementações mais exóticas podem ser concebidas a depender da estrutura de dados escolhida como representação dos genomas, contanto que o algoritmo garanta que os descendentes sejam compostos por partes herdadas de ambos ancestrais [18]. A Figura 31 exemplifica um cruzamento de dois pontos.

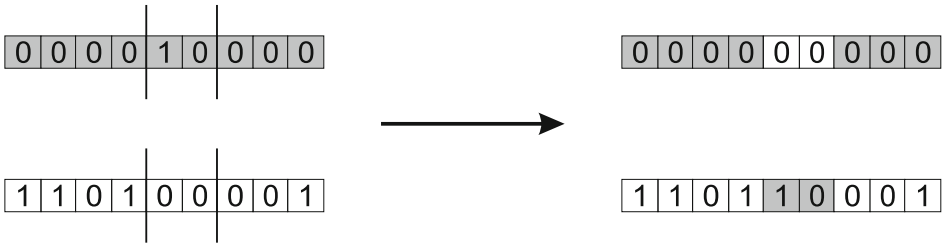


Figura 31 – Exemplo de cruzamento com dois pontos ou loci, entre genomas representados como *strings* binárias. Extraído de Eiben e Smith [11].

O uso de vários pontos reduz o viés de herança do meio de representação, ao permitir que porções menores do genoma variem individualmente. Em último caso, pode ser efetuado cruzamento uniforme, onde a chance de um indivíduo herdar material genético de um ou outro ancestral é avaliada individualmente para cada lócus do genoma, com igual probabilidade de herança para ambos os ancestrais [35].

4.3.2 Mutação

O operador de mutação é o responsável por causar pequenas alterações no genoma de um único indivíduo. Enquanto o cruzamento é responsável por permitir a busca pelo

espaço de soluções, as mutações são responsáveis por introduzir novos alelos que não evoluiriam com somente cruzamentos. Essa habilidade é útil porque permite que o algoritmo escape de ótimos locais e encontre melhores soluções. O operador de mutação σ pode ser definido, então, como na Definição 4.3 [18, 11].

$$\sigma : G \rightarrow G \tag{4.3}$$

A forma básica de mutação é implementada como uma alteração na informação codificada em algum ponto do genoma. A partir desse conceito, operadores mais sofisticados podem ser derivados, como mutação em múltiplos pontos ou mutação probabilística, onde o genoma é percorrido e cada posição possui uma probabilidade α de sofrer uma alteração [18]. A Figura 32 exemplifica um genoma que sofre mutação em três loci.

Figura 32 – Exemplo de mutação onde uma *string* binária tem três bits invertidos aleatoriamente. Extraído de Eiben e Smith [11].

4.3.3 Seleção

A seleção, ao contrário dos operadores de variação, não cria novas soluções, mas somente destaca os melhores indivíduos em uma população eliminando os menos aptos. Essa eliminação leva em conta uma avaliação da aptidão de cada solução, denominada *fitness*. A avaliação do *fitness* é a etapa que geralmente detém o maior custo computacional dos algoritmos evolutivos. De modo geral, é esperado que os indivíduos mais aptos tenham maior chance de serem selecionados, mas isso não necessariamente irá ocorrer, a depender do método de seleção utilizado [35].

Os algoritmos de seleção podem ser determinísticos, onde um número fixo dos melhores indivíduos é selecionado, ou probabilísticos, em que cada indivíduo possui uma probabilidade de seleção proporcional a seu *fitness*. Os métodos probabilísticos, ou estocásticos, são preferíveis em muitos casos, pois a estocasticidade da seleção promove a variabilidade genética da população, enquanto que os métodos determinísticos tendem a convergir prematuramente devido aos vieses da população original [35]. Dentre os algoritmos de seleção probabilísticos, destacam-se:

- **Roleta** — Na seleção por roleta, a probabilidade de cada indivíduo i ser selecionado para reprodução é igual a $\frac{fitness(i)}{\sum fitness}$, ou seja, igual à fração que corresponde o seu valor de *fitness* em relação ao somatório total da população. A Tabela 3 mostra as

probabilidades de escolha para uma população de seis indivíduos usando o método da roleta;

- **Ranque** — Semelhante à seleção por roleta, exceto por determinar a probabilidade de seleção utilizando um valor de ranqueamento ao invés do *fitness*. Os indivíduos são ordenados em ordem crescente de *fitness*, e a posição de cada indivíduo em razão do somatório de todas as posições $\frac{ranque(i)}{\sum ranque}$ é utilizada como probabilidade. É útil em casos onde a variância dos valores de *fitness* na população é muito grande ou muito pequena. As probabilidades para este método também são mostradas na Tabela 3;

- **Torneio** — Nesse método, um subconjunto de tamanho predeterminado contendo k indivíduos é escolhido aleatoriamente. Esses indivíduos competem entre si, e o vencedor, com o melhor *fitness*, é selecionado para reprodução. O tamanho k do torneio influencia a pressão seletiva: quanto maior, menor é a chance de indivíduos menos aptos serem selecionados.

Fitness	**Ranque**	P_{roleta}	P_{ranque}
2,1	1	0,099	0,048
3,6	5	0,169	0,238
7,1	6	0,333	0,286
2,4	2	0,113	0,095
3,5	4	0,164	0,190
2,6	3	0,122	0,143

Tabela 3 – Comparação das probabilidades de escolha em uma população de seis indivíduos, considerando os métodos de seleção proporcional. Adaptado de Ashlock [18].

A etapa seleção pode ser efetuada antes ou depois dos operadores de variação. Especificamente, nos paradigmas de algoritmo genético (GA) e programação genética (GP), o operador de seleção é usualmente aplicado antes da variação, de modo a selecionar boas soluções e só então efetuar recombinação e mutação. Outras formas de algoritmo evolutivo, como na programação evolutiva (EP) e estratégias evolutivas (ES), preferem aplicar os operadores de variação primeiro [35].

Nas implementações que postergam a seleção, a etapa de variação produz um conjunto de descendentes distinto da população ancestral, sobre o qual o operador de seleção é aplicado. Nesses casos, indivíduos de um ou ambos os conjuntos devem ser selecionados para compor a próxima geração do processo evolutivo, podendo ser empregadas as estratégias $(\mu+\lambda)$ ou (μ, λ), onde μ e λ são os tamanhos da população ancestral e descendente, respectivamente.

No método $(\mu + \lambda)$, são selecionadas as μ melhores soluções de um conjunto que inclui tanto os μ ancestrais quanto os λ descendentes. Como ambas as populações são

avaliadas, se realizada de forma determinística, esse esquema de seleção garante a preservação da melhor solução encontrada em qualquer iteração. Essa propriedade é denominada elitismo, e é utilizada para manter bons indivíduos sempre na população [35].

Em contrapartida, no método (μ, λ) são selecionadas as μ melhores soluções somente do conjunto de descendentes λ. Isso implica que $\lambda > \mu$ — se ambos os parâmetros populacionais forem iguais, não há seleção; o algoritmo torna-se um passeio aleatório pelo espaço de soluções — e os melhores indivíduos não tem garantia de se manterem na população. Sendo assim, essa estratégia promove uma exploração mais exaustiva quando comparada com a estratégia $(\mu + \lambda)$.

Os operadores de seleção também são caracterizados por um parâmetro denominado pressão seletiva. Ele é determinado pelo tempo de dominação da população, que representa o tempo necessário para que a solução com melhor *fitness* ocupe toda a população. Sendo assim, se a pressão seletiva é alta, a população perde diversidade rapidamente, enquanto que se é baixa, a convergência é lenta. Thierens e Goldberg [65] demonstraram uma relação entre a pressão seletiva e a probabilidade de recombinação, e que uma alta pressão seletiva deve ser acompanhada por uma alta probabilidade de recombinação para que o algoritmo evolutivo tenha sucesso [35].

4.3.4 Algoritmo Genético

O algoritmo genético (GA) é o tipo mais popular de algoritmo evolutivo, e emula de forma mais análoga os genomas reais e sua variação através do processo evolutivo. Idealizado nas décadas de 50 e 60 por Holland, Bremermann e Fraser [66, 67, 68, 69], ele é amplamente utilizado em áreas como aprendizado de máquina graças à sua habilidade de encontrar heurísticas para problemas de otimização.

Nessa modalidade de algoritmo evolutivo, os indivíduos são comumente representados na forma de *strings* binárias de tamanho fixo (Figuras 31 e 32), apesar de representações alfanuméricas também serem possíveis [70]. Essas representações, denominadas cromossomos ou genomas, são compostas por genes, tal que um gene consiste na codificação de um parâmetro do problema. Esses genes também chamados de variáveis de decisão e determinam uma ou mais características fenotípicas do indivíduo, ou pelo menos têm influência sobre elas [11]. A Tabela 4 exemplifica um cromossomo binário típico dos GAs.

variáveis	$D_1 = 22$						$D_2 = 29$					$D_3 = 28$				
bits	0	1	0	1	1	0	1	1	1	0	1	1	1	1	0	0

Tabela 4 – Cromossomo cujo conteúdo é mapeado para três variáveis de decisão D_1, D_2, D_3.

A operação de mutação atua ao aleatoriamente inverter alguns bits da *string* de representação. O cruzamento é feito simplesmente recombinando partes do cromossomo

após o particionamento em um número predeterminado de pontos. Tradicionalmente, os algoritmos genéticos aplicam seleção reprodutiva, isto é, os melhores indivíduos são selecionados antes da aplicação dos operadores de variação e, comumente, métodos estocásticos de seleção são utilizados [35].

4.3.5 Programação Genética

A programação genética (GP) é um paradigma de algoritmo evolutivo introduzido por Cramer e Koza em meados dos anos 1980 [71, 72], onde as estruturas de dados em processo adaptativo são programas de computador. Para isso, os cromossomos que correspondem aos indivíduos assumem a forma de árvores sintáticas, estruturas usadas por compiladores e interpretadores como representação interna de programas em compilação. A avaliação do *fitness* então consiste em transformar essas representações em código executável e coletar métricas de execução de cada um dos programas na população [35].

Mais especificamente, os nós das árvores sintáticas podem ser classificados como funções primitivas ou símbolos terminais. As funções primitivas correspondem a elementos do código como expressões aritméticas e de comparação, estruturas de controle e chamadas de função. Os terminais incluem elementos de valor constante ou variáveis avaliadas em tempo de execução, que servem como parâmetros do programa. Dessa maneira, as soluções estão contidas em um espaço formado por todas as possíveis composições recursivamente criadas a partir das funções primitivas e terminais disponíveis [73].

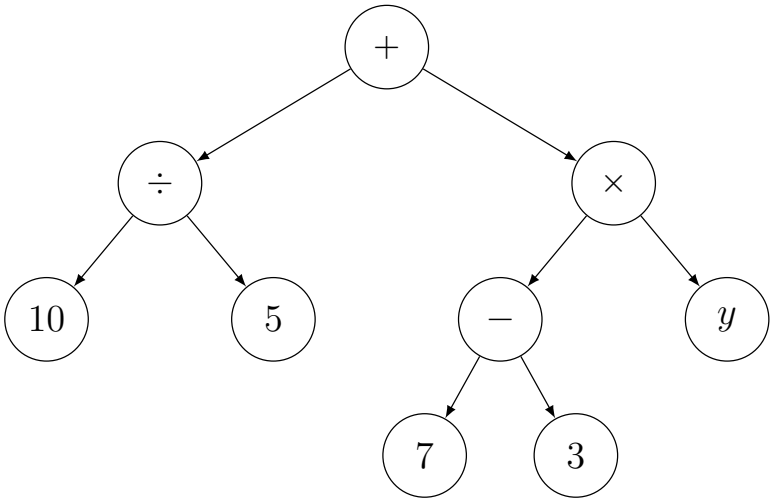


Figura 33 – Exemplo de árvore sintática com sua expressão correspondente, na comumente usada notação prefixa.

Os símbolos disponíveis para a construção de árvores sintáticas também podem carregar significado semântico. Em uma instância de programação genética de tipagem forte, por exemplo, tanto os valores de retorno de um nó quanto os argumentos das funções primitivas possuem tipo, e a estrutura da árvore sintática deve respeitar essas

restrições. Sendo assim, em uma instância fortemente tipada de programação genética onde a primitiva lógica $\vee$ possui dois operandos de tipo booleano, a expressão $1 \vee 0,45$ não configuraria um indivíduo válido no espaço de soluções, pois $0,45 \notin \{0,1\}$ [11].

A Figura 33 mostra uma árvore sintática e a tradução para a expressão correspondente em notação prefixa. Os nós folha são formados por símbolos terminais, de valor constante ou variável, enquanto os nós intermediários são funções primitivas relacionadas aos operadores aritméticos. O paradigma da programação também genética traz consigo operadores próprios de cruzamento e mutação, apropriados para manipular a estrutura de árvore dos genomas. O operador de cruzamento atua recombinando ramos das árvores ancestrais para criar dois descendentes distintos, como mostrado na Figura 34.

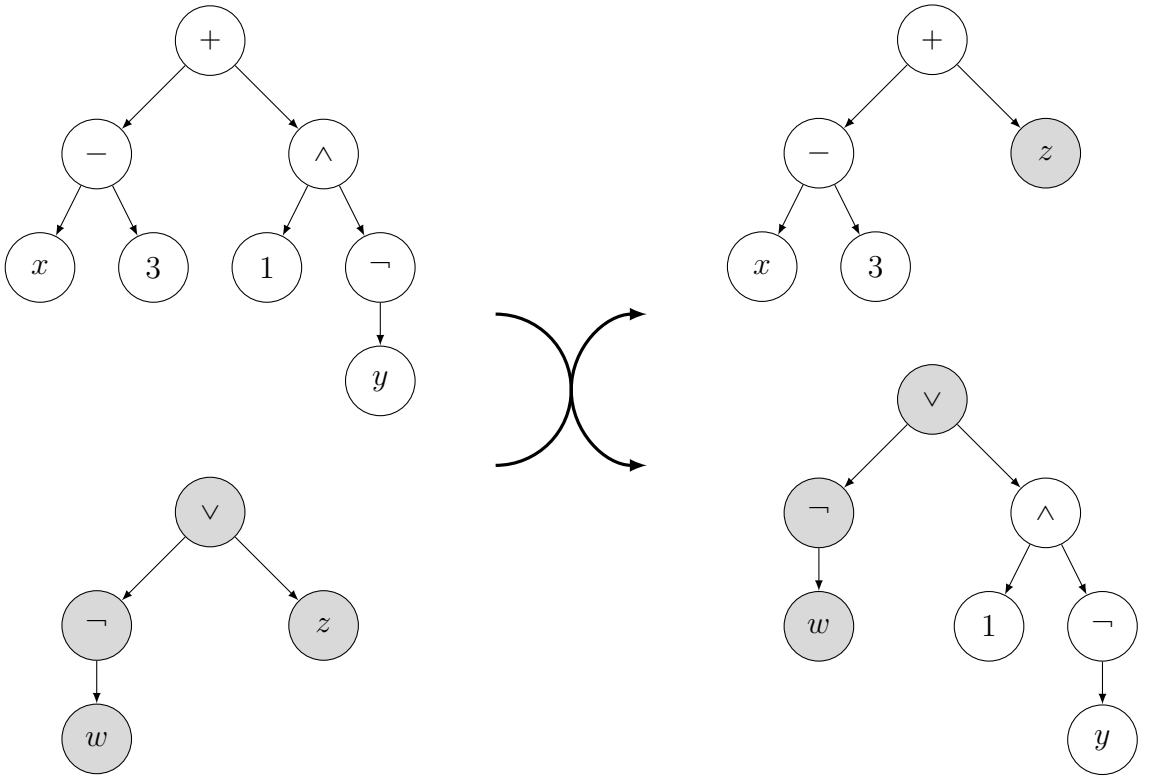


Figura 34 – Demonstração do operador de cruzamento na programação genética. As subárvores $\wedge(1, \neg(y))$ e z são recombinadas para formar os indivíduos $+(-(x, 3), z)$ e $\vee(\neg(w), \wedge(1, \neg(y)))$

Já o operador de mutação realiza alterações aleatoriamente nos nós da árvore, podendo simplesmente alterar as primitivas e valores dos terminais, bem como inserir subárvores totalmente aleatórias em algum ponto do genoma do indivíduo mutante. A Figura 35 demonstra a substituição de um nó por uma subárvore no descendente mutante. De maneira análoga aos algoritmos genéticos, a etapa de seleção é geralmente realizada antes da reprodução [73, 35].

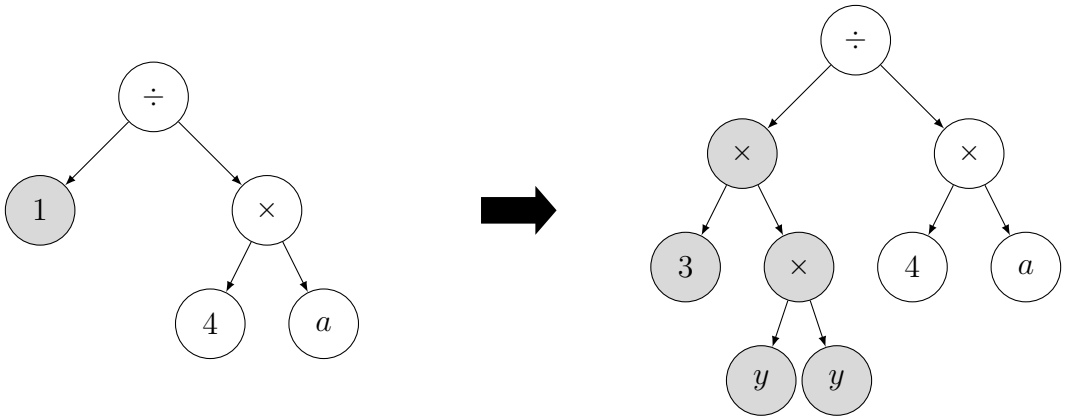


Figura 35 – Indivíduo submetido a mutação, onde o terminal 1 é substituído pela subárvore $\times(3, \times(y, y))$ aleatoriamente gerada.

4.3.6 Outros métodos

- **Estratégias evolutivas (ES)** — Introduzida por Rechenberg, Schwefel *et al.* [74, 75] nos anos 60, usa vetores de números reais como representação e tem como principal diferencial taxas de mutação variáveis. É aplicada principalmente em problemas de otimização numérica e combinatória;
- **Programação evolutiva (EP)** — Trabalha submetendo máquinas de estado finito ao processo evolutivo, buscando autômatos capazes de reconhecer determinadas cadeias de caracteres, como formalizado por Fogel *et al.* em 1966 [76];
- **Algoritmos meméticos** — Combinam elementos de algoritmos genéticos com processos de aprendizado cultural, empregando métodos evolutivos para encontrar novas soluções e refinamentos locais para melhorar as soluções encontradas [77].

5 TRABALHOS CORRELATOS

Com a recente expansão da pesquisa em torno da inteligência artificial, uma variedade de trabalhos foram realizados buscando integrá-la 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 *data-sets* para treinamento e a necessidade de exatidão nos algoritmos de geração de código, contrastando com a natureza aproximada das soluções produzidas por modelos de IA [15].

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

5.1 Minimização de *Spill Code* com Algoritmos Evolutivos

Em 2003, Stephenson *et al.* [12] publicaram um trabalho que discute o uso de técnicas de inteligência artificial para obter automaticamente as heurísticas de várias etapas de otimização, incluindo a alocação de registradores. Os autores introduzem o conceito de *"Meta Optimization"* ou metaotimização, em uma tradução livre, uma metodologia que emprega algoritmos evolutivos 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 se obter boas heurísticas, representadas na forma de expressões aritméticas avaliáveis em tempo de compilação, através do processo evolutivo.

Na abordagem de Stephenson *et al.* [12], as melhores heurísticas foram obtidas através do processo de evolução, sendo elas representadas como árvores sintáticas de expressões aritméticas tais quais as mostradas na Figura 36. O algoritmo de alocação de registradores usado nos experimentos foi o de alocação prioritária, onde a prioridade representa o ganho de performance ao se manter um *live range* em registradores, ao invés de se realizar *spill*.

As heurísticas seriam usadas para determinar essa prioridade, influenciando na

qualidade do resultado final da alocação. As árvores eram compostas de funções primitivas, como operações aritméticas entre números reais, e terminais que incluiam constantes e parâmetros sobre o *live range* tendo sua prioridade calculada, como número de usos, definições, execuções, dentre outros.

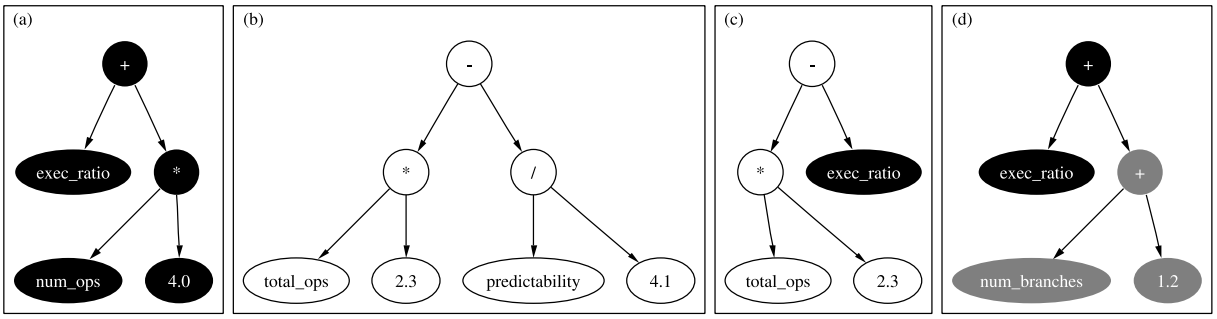


Figura 36 – Cromossomos das funções de prioridade, submetidas a operações de cruzamento e mutação. Extraído de Stephenson *et al.* [12].

O *fitness* das heurísticas foi avaliado usando-as para compilar um conjunto de *benchmarks*, coletando o tempo de execução dos programas produzidos por cada heurística e calculando o ganho de velocidade em comparação à heurística prioritária original. Uma população de 400 heurísticas foi submetida ao processo evolutivo por 50 gerações, para o qual houve uma taxa de substituição geracional média de 22% dos indivíduos. Foi aplicada uma taxa de mutação de 5%, e o algoritmo de seleção empregado foi o de seleção por torneio, utilizando-se um torneio de tamanho 7 [78].

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 significativo. A Figura 37 mostra o *plot* do nível de *fitness* ao longo das gerações, para o treinamento das heurísticas especializadas e da heurística de uso geral, respectivamente.

Outros trabalhos, como os de Kri e Feeley (2004) [79], Mahajan e Ali (2008) [80], além de Topcuoglu *et al.* (2007) [81], realizam experimentos com alocadores inteiramente genéticos, onde o mapeamento dos registradores virtuais para os físicos é representado na forma de cromossomos. Uma alocação válida para o conjunto de variáveis é então obtida através de um processo evolutivo, ao invés dos algoritmos tradicionais.

5.2 Coloração de Grafos via *Deep Learning*

Em um trabalho de 2019, Lemos *et al.* [82] lidam com problema da coloração utilizando um modelo de rede neural em grafo (GNN), um tipo de rede neural capaz

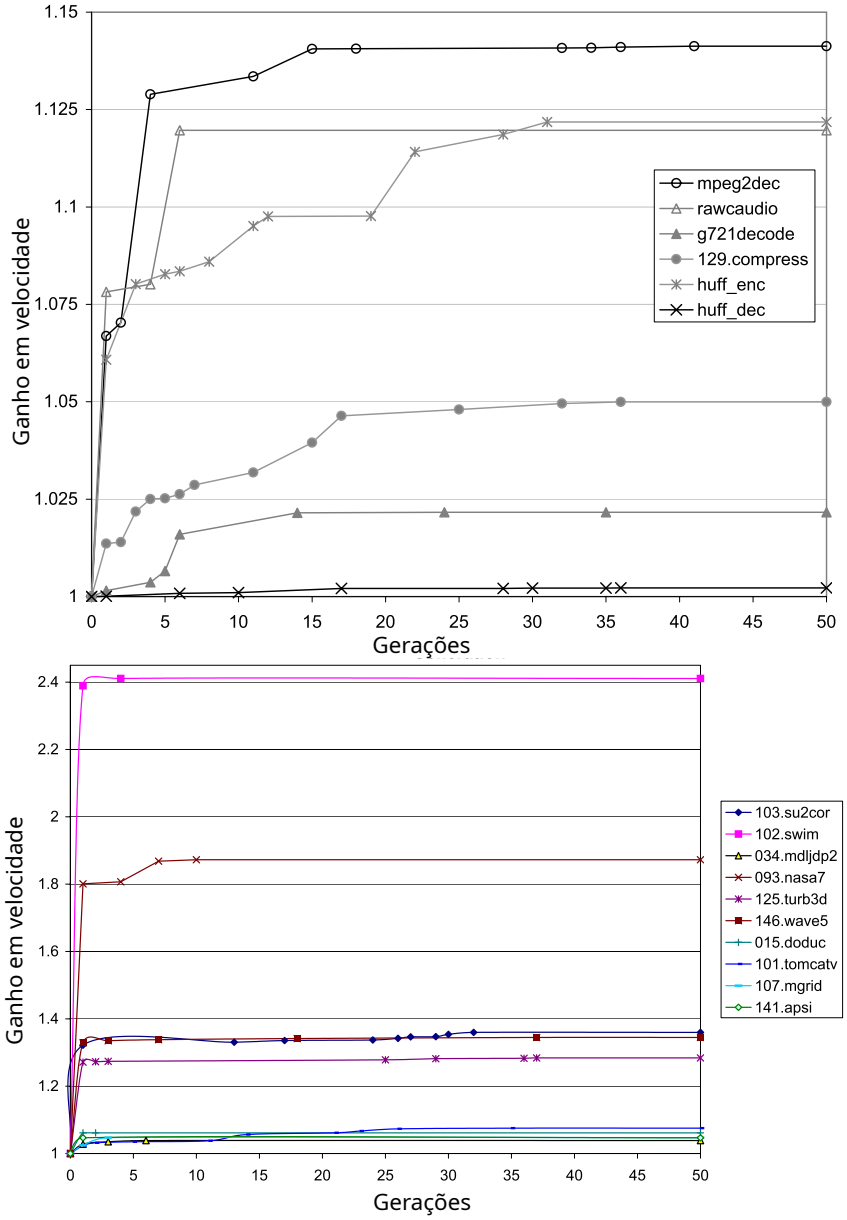


Figura 37 – O eixo horizontal mostra o número de gerações, enquanto o eixo vertical apresenta o aumento na velocidade de execução. Extraído de Stephenson *et al* [12].

de representar informações extraídas dos nós de um grafo [83]. 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 con-

junto 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.* [14] 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 38 ilustra a organização básica de uma célula LSTM.

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

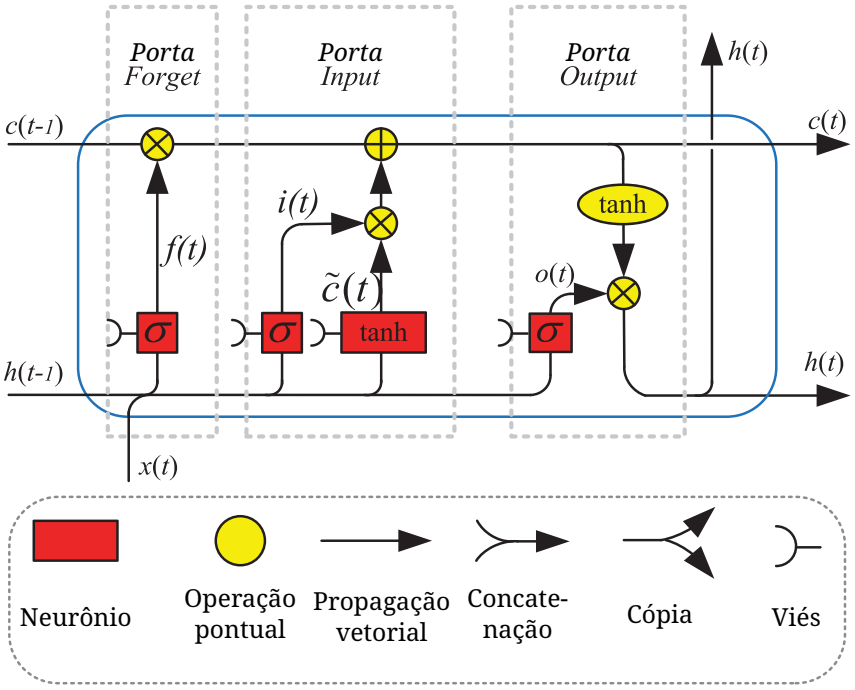


Figura 38 – Célula de memória LSTM. Extraído de Yu *et al.* [13].

futuros [13, 14].

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 39 esquematiza a organização do modelo.

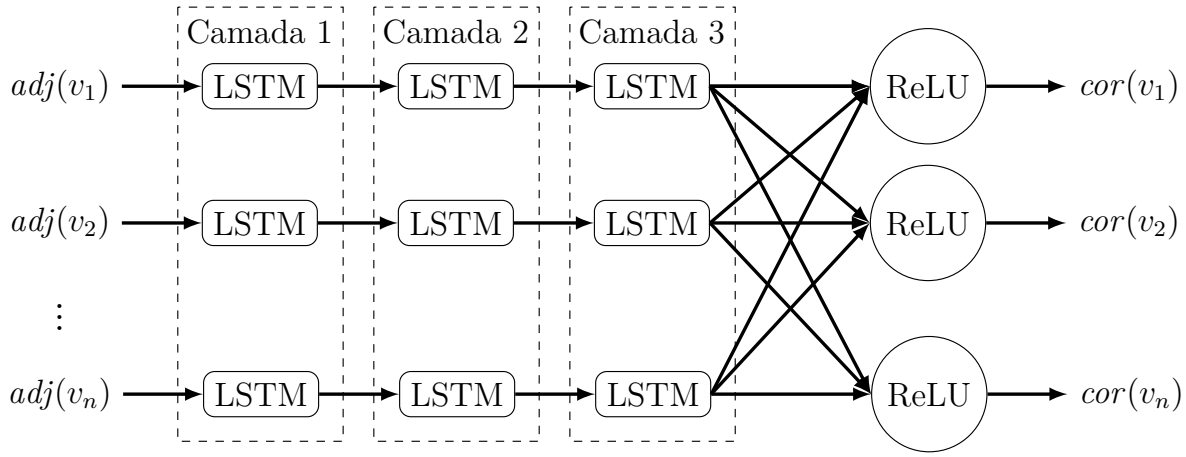


Figura 39 – Esquema da rede neural de coloração. Adaptado de Das *et al.* [14].

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 [84].

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.* [15] 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.* [82] (5.2), o modelo de inteligência artificial de VenktaKeerthy *et al.* [15] 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 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 40 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 [16]. A Figura 41 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.

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

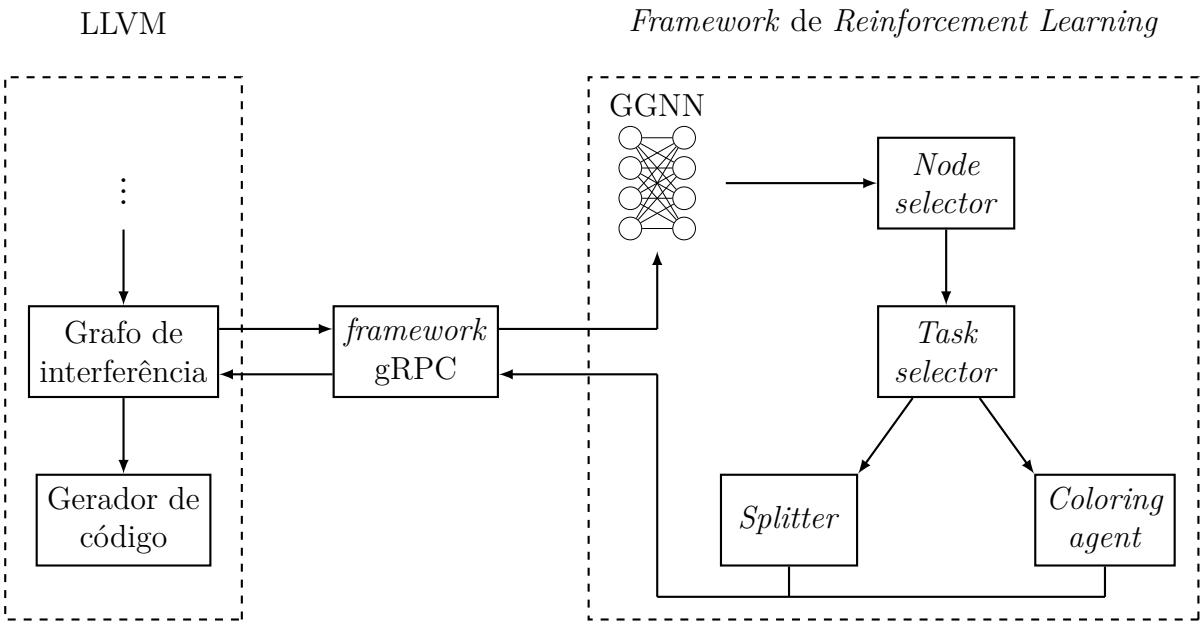


Figura 40 – Esquema da interação entre o LLVM e o *framework* de *RL*. Adaptado de VenkataKeerthy *et al.* [15].

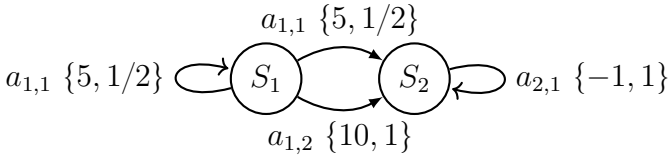


Figura 41 – Esquema de uma instância de processo de decisão de Markov. Adaptado de Puterman [16].

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.* [85] 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.* [86], por exemplo, desenvolveram um método de coloração de grafos com redes neurais inspirado por princípios da física. Goudet *et al.* [87] propuseram um modelo combinando *deep learning* e um *framework* memético, um tipo de algoritmo evolutivo, executável em GPUs.

Dodaro *et al.* [88] 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 [89] identificaram 78 características dos grafos que podem indicar quais algoritmos se utilizar para realizar a coloração, e sugeriram técnicas de aprendi-

zado de máquina para automaticamente classificar grafos e 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 inteligência artificial.

6 PROPOSTA DE ALOCADOR DE REGISTRADORES TREINADO COM PROGRAMAÇÃO GENÉTICA

Inspirado pela implementação de Stephenson [12, 78], este trabalho se propõe a implementar um alocador de registradores que faça uso de heurísticas obtidas através de algoritmos evolutivos. Mais especificamente, almeja-se reproduzir, mesmo que parcialmente, os resultados obtidos por Stephenson, que empregou programação genética no treinamento de heurísticas para seu alocador de registradores prioritário.

Stephenson [12, 78] demonstrou que a computação evolutiva é um boa abordagem de inteligência artificial para ser aplicada no desenvolvimento de compiladores. O treinamento através do processo evolutivo se mostrou capaz de encontrar eficientes heurísticas de uso específico e geral. Em particular, o trabalho de Stephenson mostra como os algoritmos evolutivos podem oferecer soluções quando não há disponibilidade de grandes conjuntos de dados e as relações entre as variáveis do problema são pouco compreendidas.

O trabalho de Bernstein [29] também demonstra como diferentes programas se beneficiam da utilização de heurísticas distintas, havendo espaço para uma otimização mais individual e específica que leva em conta as características do código. Essa constatação também motiva uma abordagem que busca o ajuste fino de determinadas heurísticas para cada categoria de programa. Sendo assim, neste trabalho as heurísticas de Bernstein serão utilizadas como parâmetro para avaliar as modificações feitas no compilador.

O LLVM é uma coleção de módulos e ferramentas de compilação utilizada por desenvolvedores de compiladores devido à sua versatilidade e modularidade. Seu funcionamento se baseia no uso de uma representação intermediária (IR) própria, mostrada na Figura 42, que funciona como uma linguagem de montagem abstrata. Ela é utilizada pelo *back-end* LLVM para realizar otimizações e gerar código para uma variedade de arquiteturas-alvo [36, 90, 84].

O *framework* do LLVM provê ferramentas para manipulação da representação intermediária e geração de código, além de *front-ends* que compilam de linguagem de alto nível para IR, como o compilador Clang, de C/C++. Uma vez possuindo a IR de um programa, código executável pode ser gerado de maneira autônoma através do *back-end* do LLVM [36]. Por isso, o LLVM será utilizado para implementar as técnicas propostas e obter resultados experimentais.

Para a implementação do modelo de treinamento utilizando algoritmos evolutivos a biblioteca DEAP [91], em Python, mostra-se útil ao prover uma gama de variedades de algoritmos evolutivos, como a programação genética, além de ferramentas para coleta de estatísticas e paralelização de execução. Será implementado um modelo de programa-

```
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"Hello, ␣
   World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0
    {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  %6 = call i32 (ptr, ...) @printf(ptr noundef @.str)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #1
```

Figura 42 – Exemplo de representação intermediária (IR) do LLVM para um programa do tipo "*Hello, World!*". Os números precedidos por "%" são registradores virtuais.

ção genética que irá manipular uma população de heurísticas candidatas, que terão seu *fitness* avaliado inserindo-as no compilador modificado e medindo o tempo de execução dos programas gerados. O código será portado para um ambiente de experimentos onde a execução ocorrerá de maneira dedicada.

Como material de treinamento e validação, propõe-se a utilização dos arquivos-fonte dos *benchmarks* do SPEC CPU® 2017. O SPEC traz amostras de *software* real utilizado em pesquisa acadêmica e produção, junto a ferramentas para coleta de métricas de execução e avaliação de performance. Vários desses programas realizam tarefas de computação intensiva, como simulações de sistemas complexos e compressão de dados, que produzem grande pressão sobre os registradores do ambiente de execução [19]. Por ser um padrão da indústria e pelas características dos programas de sua coletânea de *benchmarks*, o SPEC foi escolhido para a realização de experimentos neste trabalho.

Os detalhes das técnicas utilizadas e decisões de implementação serão explicados nas seções a seguir. Os resultados obtidos a partir dos experimentos serão discutidos no Capítulo 7.

6.1 Compilador LLVM Modificado

Na etapa de alocação de registradores, o LLVM possui quatro implementações distintas de alocadores: `fast`, `basic`, `greedy` e `pbqp`. Dos quatro, `fast`, `basic` e `greedy` realizam alocação via *linear scan*, enquanto `pbqp` é um alocador experimental que utiliza a abordagem via PBQP. O alocador `fast` realiza a forma mais simples de *linear scan*, percorrendo os registradores virtuais de maneira sequencial localmente nos blocos básicos, e inserindo *spill code* somente no final dos blocos [92], enquanto o `pbqp` utiliza um método de alocação muito diferente dos demais. Por isso, neste trabalho o enfoque maior será nos alocadores de *linear scan* `fast` e `greedy`.

O alocador `basic` implementa uma extensão do método de *linear scan* de Poletto e Sarkar [2], onde os registradores são alocados em um único passe pelo código e os *live intervals* são percorridos em uma ordem predeterminada. No entanto, ao invés de um percurso linear ou em pós-ordem pelo grafo de controle de fluxo, como em implementações tradicionais, o alocador `basic` segue uma ordem prioritária. Cada registrador virtual tem seu peso de *spill* calculado em um passe anterior, e esse peso é usado como valor de prioridade durante a alocação [92].

O alocador `greedy` é semelhante ao `basic`, funcionando com um *linear scan* prioritário. No entanto, o critério de prioridade adotado pelo alocador `greedy` é o tamanho dos *live intervals*. A utilização do peso, que representa a densidade de usos normalizada em relação ao tamanho da variável, promove a alocação dos menores intervalos vivos primeiro. Isso comumente causa o *spill* de grandes intervalos, justificando a mudança na heurística usada. Além disso, o alocador `greedy` utiliza técnicas mais sofisticadas para minimizar a geração de *spill code*, como o *live range splitting* antes da alocação e a possibilidade de desalocar registradores físicos *on the fly* (*eviction*) [92].

Em ambos `basic` e `greedy`, a lista de intervalos é armazenada como uma fila de prioridade, que usa a implementação disponibilizada pela biblioteca padrão do C++. Sendo assim, o código-fonte foi modificado para que a fila de prioridade use como valor de prioridade o resultado de uma expressão, avaliada em tempo de compilação e recebida como argumento na chamada do compilador através da linha de comando. Essa expressão é a forma textual das heurísticas candidatas, que podem ser livremente alteradas conforme a necessidade.

A Figura 43 mostra o formato textual das expressões aceitas pelo compilador modificado junto às respectivas representações em árvore. Esse formato é o mesmo utilizado pela biblioteca DEAP para a manipulação dos cromossomos no algoritmo de programação genética. Nesta implementação, foi utilizado uma variante fortemente tipada de programação genética. Sendo assim, as expressões estão restritas às combinações válidas de acordo com o tipo de retorno de cada primitiva ou terminal, tal que os símbolos das árvores

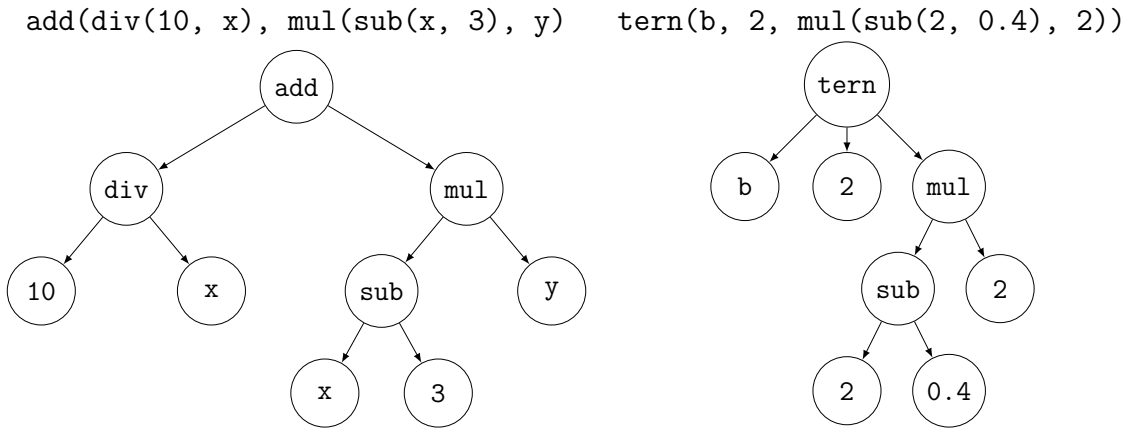


Figura 43 – Exemplos de expressões no formato aceito pelo *parser* de funções heurísticas embutido no LLVM, e suas respectivas representações em árvore.

sintáticas podem possuir valor real (R) ou booleano (B).

As primitivas reconhecidas pelo compilador incluem operações de retorno real, como as listadas na Tabela 5, ou operações lógicas e de comparação, apresentadas na Tabela 6. Os símbolos terminais incluem constantes numéricas de valor real ou booleano, além de um conjunto de variáveis cujo valor é obtido através das análises de código do LLVM.

Representação	**Operação**
`add(`$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$R_1 + R_2$
`sub(`$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$R_1 - R_2$
`mul(`$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$R_1 \times R_2$
`div(`$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$R_1 \div R_2$
`pow(`$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$R_1^{R_2}$
`sqrt(`$\mathtt{R}_1$`)`	$\sqrt{R_1}$
`tern(`$\mathtt{B}_1$`, `$\mathtt{R}_1$`, `$\mathtt{R}_2$`)`	$\begin{cases} R_1, & \text{se } B_1 \\ R_2, & \text{caso contrário} \end{cases}$

Tabela 5 – Representação textual das primitivas de retorno real reconhecidas pelo compilador, junto às respectivas definições matemáticas, nas quais R são valores reais e B são valores lógicos.

As variáveis terminais contém informações sobre os registradores virtuais e seus respectivos *live intervals* associados, constituindo os parâmetros que produzem o espaço de soluções investigado pelo algoritmo evolutivo. A escolha das variáveis foi inspirada pela implementação de Stephenson [78] e pautada pelas estatísticas disponibilizadas pelo LLVM. As variáveis de retorno real são listadas na Tabela 7, enquanto as de retorno booleano na Tabela 8.

Representação	Operação
`and(B1, B2)`	$B_1 \wedge B_2$
`or(B1, B2)`	$B_1 \vee B_2$
`not(B1)`	$\neg B_1$
`lt(R1, R2)`	$R_1 < R_2$
`gt(R1, R2)`	$R_1 > R_2$
`eq(R1, R2)`	$R_1 = R_2$

Tabela 6 – Representação textual das primitivas de retorno booleano reconhecidas pelo compilador, junto às respectivas definições matemáticas, nas quais R são valores reais e B são valores lógicos.

Variável	Parâmetro
`original`	Heurística original do alocador selecionado
`cost`	Custo de *spill* de Chaitin [1] e Bernstein [29]
`degree`	Número de interferências do registrador virtual
`area`	Área do registrador virtual de Bernstein [29]
`instructions`	Número de instruções onde o registrador virtual é vivo
`uses`	Número de usos do registrador virtual
`defs`	Número de definições do registrador virtual
`calls`	Número de chamadas de função onde o registrador virtual é vivo
`refs`	Número de referências à endereços de memória onde o registrador virtual é vivo
`moves`	Número de *moves* do registrador virtual
`averageFreq`	Frequência média de execução do registrador virtual
`numValues`	Número de valores no intervalo associado ao registrador virtual
`size`	Tamanho do intervalo associado ao registrador virtual
`numBlocks`	Número de blocos básicos onde o registrador virtual é vivo

Tabela 7 – Variáveis contendo informações sobre os registradores virtuais de retorno real.

6.2 Modelo de Programação Genética

A cada geração do modelo de programação genética, os cromossomos da população são convertidos para a sua forma textual e inseridos no compilador, servindo como funções heurísticas para compilar um *benchmark* de treinamento. Então, é medido o tempo de execução dos executáveis gerados por cada cromossomo, e esse tempo é usado como o valor de *fitness*. A variação escolhida de algoritmo evolutivo foi modificada para realizar uma minimização do *fitness*, ao invés da maximização tradicional, para que sejam encontradas heurísticas que reduzam o tempo de execução.

A escolha dos parâmetros populacionais para o algoritmo de programação genética também se baseou nos experimentos de Stephenson, mas foi influenciada pelas limitações do ambiente de testes. A coleta do tempo de execução para cada heurística se mostrou um processo demorado e os recursos para efetuar um grande número de medições em para-

Variável	Parâmetro
`isSpillable`	Se o intervalo associado ao registrador virtual pode sofrer *spill*
`hasAtLeastOneValue`	Se o intervalo associado ao registrador virtual possui ao menos um valor ao longo do programa
`isTerminator`	Se o intervalo associado ao registrador virtual se encontra no conjunto de saída de algum bloco básico

Tabela 8 – Variáveis contendo informações sobre os registradores virtuais de retorno booleano.

lelo, acelerando a execução do algoritmo, não estavam disponíveis. Portanto, foi definido um tamanho populacional de 100 indivíduos e o processo evolutivo foi efetuado por 30 gerações, valores inferiores aos usados por Stephenson [12, 78].

A probabilidade de cruzamento foi mantida em 90%, aplicando recombinação de subárvores em um único ponto aleatoriamente escolhido em ambos os indivíduos. A mutação foi realizada com uma probabilidade de 10%, também em um único ponto aleatório, inserindo novas expressões no lugar da subárvore com raiz no ponto de mutação. O algoritmo de variação, provido pela biblioteca, aplica exclusivamente um dos dois métodos por indivíduo, nunca ambos.

A seleção foi realizada de maneira probabilística através do método de seleção por torneio, com torneio de tamanho 7 efetuado em uma única rodada. A cada geração, a seleção foi aplicada sobre o conjunto contendo tanto os indivíduos ancestrais quanto os descendentes, tal como na estratégia evolutiva $(\mu + \lambda)$. Esse tipo de seleção propicia um grau considerável de elitismo, pois indivíduos aptos tem uma alta probabilidade de se manterem na população, mas ao mesmo tempo permite que a variabilidade genética se mantenha alta por mais iterações devido à seleção estocástica.

Parâmetro	Valor
População	100
Gerações	30
Estratégia	$(\mu + \lambda)$
μ	100
λ	100
Seleção	Torneio, $n = 7$
$P_{cruzamento}$	90%
$P_{mutação}$	10%

Tabela 9 – Parâmetros de execução do modelo de programação genética.

Também foi utilizado um *hall-of-fame*, uma estrutura para armazenar o melhor indivíduo encontrado ao longo de todo o processo evolutivo, onde a melhor solução estaria armazenada ao final do treinamento. A Tabela 9 sumariza os parâmetros de execução do modelo de programação genética utilizado. Também foi aplicado ao algoritmo um limite de

altura de 17 níveis para as árvores correspondentes aos cromossomos pois, na programação genética, os cromossomos tendem se tornar excessivamente grandes.

6.3 *Benchmarks* de Treinamento e Ambiente de Execução

Para mensurar os efeitos das modificações no compilador e o desempenho das heurísticas candidatas foram utilizados os programas da coletânea do SPEC CPU® 2017. A *suite* do SPEC inclui programas das mais variadas aplicações na computação científica e de produção, escritos em linguagem C, C++ e Fortran. Na realização dos testes, os arquivos foram compilados para a representação intermediária do LLVM através dos compiladores Clang, de C/C++, e Flang, de Fortran.

Dos 43 *benchmarks*, 39 compilaram com sucesso. Alguns programas que possuem trechos de código Fortran não foram compilados corretamente pelo Flang, que ainda é um compilador experimental e em desenvolvimento ativo. Ainda assim, o Flang foi capaz de gerar executáveis válidos para os outros *benchmarks* em Fortran. Os arquivos foram compilados usando o perfil de otimização *peak* do SPEC, que conta com *flags* específicas, e o compilador *back-end* do LLVM `llc` foi chamado com a *flag* `-O3`.

Em posse dos arquivos de IR, o *back-end* modificado do LLVM foi acionado através do executável `llc` para produzir arquivos executáveis e coletar estatísticas sobre o código produzido usando as heurísticas candidatas. As heurísticas de Bernstein [29] foram usadas para verificar o impacto da alteração na heurística de prioridade sobre a geração de código *spill*. Os arquivos do SPEC forma compilados para as arquiteturas x86-64 e AArch64 com as heurísticas de Bernstein, sendo coletados os totais de instruções de `store` e `load`.

Benchmark	**Área de aplicação**
`505.mcf_r`	Planejamento de rota de veículos
`508.namd_r`	Simulação de sistemas biomoleculares
`525.x264_r`	*Encoding* de vídeo
`531.deepsjeng_r`	IA para xadrez usando poda alfa-beta
`541.leela_r`	IA para *Go* usando árvores de Monte Carlo

Tabela 10 – Benchmarks selecionados para treinamento e validação do modelo de programação genética e suas respectivas aplicações [19].

Foram aleatoriamente selecionados cinco *benchmarks*, mostrados na Tabela 10, para serem usados como elementos de treinamento e validação do modelo de programação genética. Os treinamentos foram realizados em uma máquina Linux x86-64, Intel Core® i3-4170, 2 núcleos com 4GB RAM, durando aproximadamente uma semana por resultado. Cada indivíduo foi executado dez vezes e o valor do *fitness* foi calculado como a média de tempo das execuções, sendo que as avaliações foram feitas dois a dois em paralelo.

Para o treinamento foram usados os *inputs* da *workload* `train` do SPEC, por seu tempo de execução menor, enquanto a validação usou a *workload* `ref`[1]. O Algoritmo 4 sumariza o funcionamento do modelo de programação genética, enquanto a Figura 44 esquematiza o método para avaliação do *fitness* dos indivíduos, que integra a implementação em Python ao *back-end* do compilador LLVM modificado.

Algoritmo 4 Funcionamento do modelo proposto para a obtenção automática de funções heurísticas via programação genética.

$P \leftarrow$ 100 funções heurísticas (heurística original, heurísticas de Bernstein e aleatórias)
$\mathit{HallOfFame} \leftarrow \{\}$
CalcularFitness(P)
repita
  $P' \leftarrow$ CruzamentoOuMutação(P, 90%, 10%)
  CalcularFitness(P')
  $P \leftarrow$ SeleçãoPorTorneio($P \cup P'$, 7)
  $\mathit{HallOfFame} \leftarrow \max\{\mathit{fitness}[i] : i \in P \cup \mathit{HallOfFame}\}$
até 30 gerações

procedimento CalcularFitness(P)
  para todo indivíduo $i \in P$ **faça**
    compilar *benchmark* usando forma textual de i
    medir 10 vezes o tempo de execução do *benchmark* compilado
    $\mathit{fitness}[i] \leftarrow$ média dos tempos de execução medidos

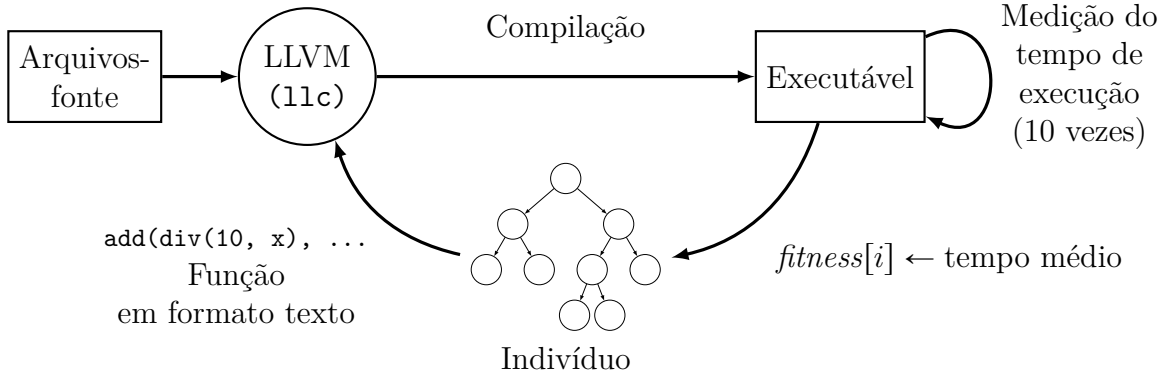


Figura 44 – Esquematização do método de avaliação do *fitness*, aplicado a todos os indivíduos ao longo das gerações do algoritmo de programação genética.

Para coletar o número de acessos à memória em tempo de execução foi utilizado o Intel® Pin, uma ferramenta de instrumentação e análise de código que funciona realizando a injeção de código executável em um programa em tempo de execução [93]. Código foi inserido nos *benchmarks* compilados com as heurísticas obtidas de modo a contar as execuções de instruções de *read* e *write* em memória.

[1] As *workloads* diferem pelo tamanho e quantidade de computações desencadeadas pelos arquivos de entrada dos *benchmarks*. A documentação oficial recomenda a utilização da *workload* `ref` para a coleta de métricas, mas os inputs da `train` também podem ser apropriados se a duração dos experimentos for um limitador [19].

7 RESULTADOS

Os resultados da compilação dos *benchmarks* usando a heurística original e as três heurísticas de Bernstein, h_1, h_2 e h_3, são mostrados nas Tabelas 11 (`basic`, x86-64), 12 (`greedy`, x86-64), 13 (`basic`, AArch64) e 14 (`greedy`, AArch64). O compilador realizou a contagem de todas as instruções de acesso à memória após a alocação de registradores, e em ambas as arquiteturas x86-64 e AArch64 foram observadas diferenças significativas na quantidade de `stores` e `loads`, a depender da heurística escolhida.

Nos *benchmarks* compilados com o alocador `basic`, se sobressaiu a heurística h_1. Em 30 programas, na arquitetura x86-64, e 34, na arquitetura AArch64, a heurística h_1 foi a que produziu o menor total de instruções de acesso à memória no binário final. A heurística h_2 foi melhor em 5 e 4 programas, em x86-64 e AArch64 respectivamente; h_3 foi melhor em 3 programas e a heurística original em 1 programa, todos na arquitetura x86-64 somente.

Benchmark	#store				#load			
	Original	h_1	h_2	h_3	**Original**	h_1	h_2	h_3
502.gcc_r	24378	21934	22664	23469	82207	66096	65815	66881
510.parest_r	55025	52288	54419	54933	125825	116265	120251	120452
526.blender_r	38528	36498	37462	37916	105701	93867	96400	97284
538.imagick_r	9094	8606	8588	8702	29733	25075	28001	26747
554.roms_r	15153	12220	12433	12455	32845	28524	29495	29433
Média (geral)	9838,5	8943,29	9094,26	9219,21	25089,32	22703,29	23283,84	23414,97

Tabela 11 – Total de instruções de `store` e `load` contabilizadas no código resultante do processo de alocação de registradores, usando o alocador `basic` na arquitetura x86-64.

Benchmark	#store				#load			
	Original	h_1	h_2	h_3	**Original**	h_1	h_2	h_3
502.gcc_r	17146	16924	17558	17236	37811	38876	41621	41486
510.parest_r	45231	45575	47246	46275	78780	76428	77201	77292
526.blender_r	31796	31583	31957	31908	54487	54161	57133	57282
538.imagick_r	6911	7163	7327	7455	17032	17925	19547	19511
554.roms_r	9868	9890	9845	9830	19104	19434	19261	19200
Média (geral)	7652,39	7482,82	7612,16	7554,97	13912,21	13758,58	14566,79	14535,66

Tabela 12 – Total de instruções de `store` e `load` contabilizadas no código resultante do processo de alocação de registradores, usando o alocador `greedy` na arquitetura x86-64.

Já no caso do alocador `greedy`, constata-se que a disparidade entre as heurísticas, particularmente da original em relação às três heurísticas de Bernstein, é substancialmente menor. Na verdade, a heurística original foi a que obteve o melhor desempenho com o alocador `greedy`, se sobressaindo em 18 programas de x86-64 e 22 de AArch64. h_1 foi

Benchmark	#store				#load			
	Original	h_1	h_2	h_3	**Original**	h_1	h_2	h_3
502.gcc_r	14116	11317	11899	12303	46639	34896	35095	36310
510.parest_r	51449	50350	51415	51976	131095	120668	124657	125809
526.blender_r	16328	14254	15011	15353	62622	48961	51867	52866
538.imagick_r	4035	3963	4139	4242	18550	16293	15496	15787
554.roms_r	7047	5538	5665	5794	31024	28435	29053	29445
Média (geral)	5144,43	4643,45	4805,57	4886,19	16506,86	13964,62	14413,21	14615

Tabela 13 – Total de instruções de `store` e `load` contabilizadas no código resultante do processo de alocação de registradores, usando o alocador `basic` na arquitetura AArch64.

Benchmark	#store				#load			
	Original	h_1	h_2	h_3	**Original**	h_1	h_2	h_3
502.gcc_r	8184	7917	8112	7887	19149	19368	20583	20317
510.parest_r	48207	51005	52904	53072	89743	86056	88603	88115
526.blender_r	12458	12406	12535	12404	25641	25910	27293	27167
538.imagick_r	3023	2933	3203	3263	8915	9038	9793	11438
554.roms_r	5655	5627	5625	5603	18005	18612	19029	19347
Média (geral)	4159,48	4206,33	4277	4258,93	8711,67	8723,12	9008,17	9062,98

Tabela 14 – Total de instruções de `store` e `load` contabilizadas no código resultante do processo de alocação de registradores, usando o alocador `greedy` na arquitetura AArch64.

superior em 11 programas tanto de x86-64 quanto de AArch64; h_2 foi superior em 4 e 2, enquanto h_3 se sobressaiu em 6 e 3 programas de x86-64 e AArch64 respectivamente.

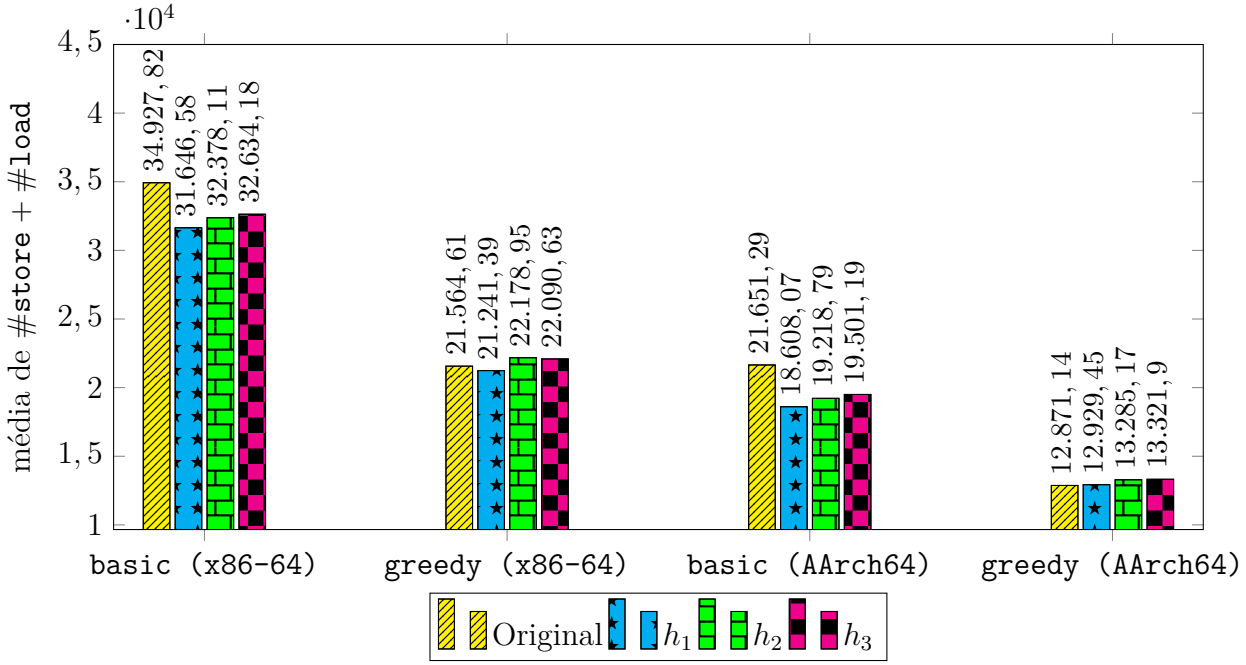


Figura 45 – Comparativo da média de *spill code* após alocação de registradores, entre alocadores e arquiteturas.

Independente da arquitetura ou da heurística escolhida, o alocador `greedy` produziu menos *spill code* para todos os *benchmarks*, exceto pelo programa `519.lbm_r`. Isso é

um reflexo da implementação de técnicas mais sofisticadas de minimização de *spill code* no alocador `greedy`, como o *live range splitting* e a capacidade de se fazer *spill* de um registrador físico já alocado (*eviction*). A Figura 45 apresenta uma comparação gráfica entre as médias *spill code* por alocador e arquitetura.

Também nota-se que, por depender somente da heurística de prioridade para minimizar a inserção de *spill code*, o alocador `basic` é mais sensível às alterações na função heurística. Entretanto, na maioria dos casos os programas gerados com o alocador `greedy` performam melhor [92, 15], e o `greedy` é a escolha padrão quando o nível máximo de otimização é habilitado no compilador. Por isso, os treinamentos com o algoritmo de programação genética se concentraram nele.

O modelo de programação genético foi executado duas vezes, uma vez utilizando o *benchmark* `531.deepsjeng_r` e outra `508.namd_r` como elementos de treinamento em cada execução, ao longo de 11 e 5 dias respectivamente. A Figura 46 mostra de maneira gráfica o *fitness* — melhora no tempo de execução em relação à geração inicial — ao longo das iterações do algoritmo.

Para o programa `531.deepsjeng_r`, o modelo rapidamente encontrou um ótimo local por volta da 10ª geração e somente pequenos incrementos no tempo de execução foram obtidos até a geração 30, convergindo para um tempo de execução de aproximadamente 49 segundos com a *workload* de treinamento. Ao final da execução foi encontrada a heurística mostrada à esquerda na Figura 47.

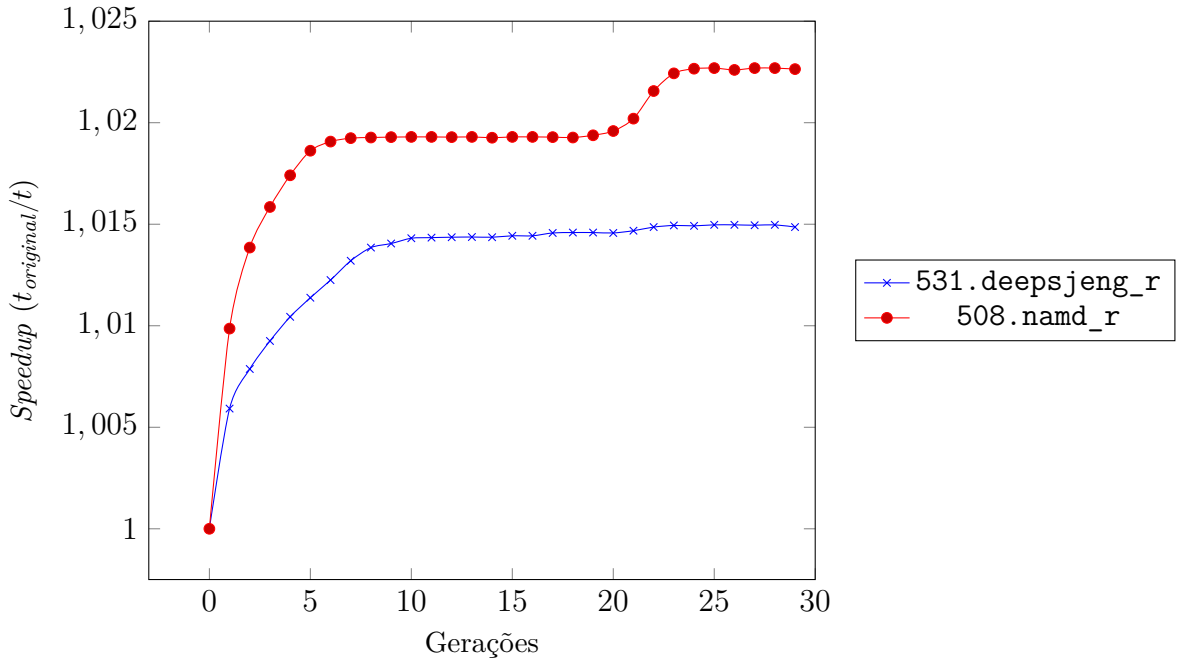


Figura 46 – *Speedup* médio dos indivíduos da população ao longo das gerações do processo evolutivo.

A função heurística à esquerda na Figura 47 foi então avaliada para um conjunto

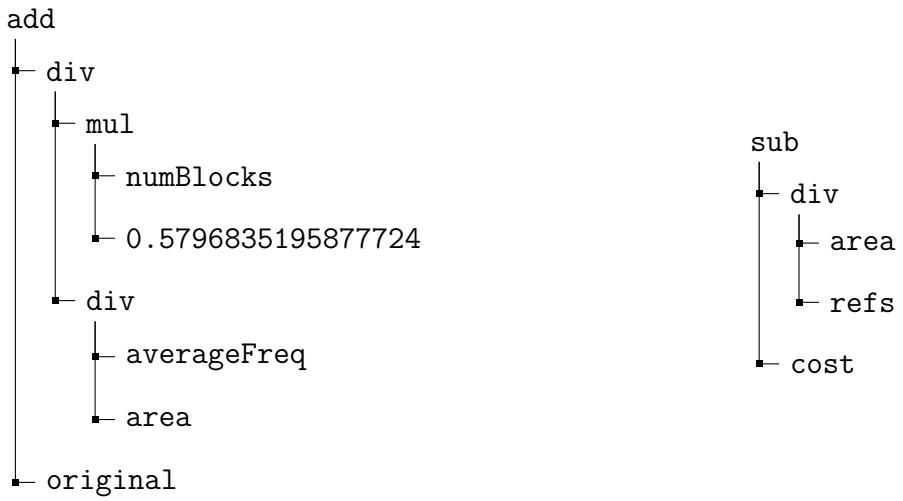


Figura 47 – Representação hierárquica das melhores heurísticas encontradas pelo modelo de programação genética, na esquerda sendo realizado o treinamento com o programa `531.deepsjeng_r`, e na direita com o programa `508.namd_r`.

de *benchmarks* usando a *workload* `ref` do SPEC, de modo a medir seu impacto na execução dos programas. O resultado dessas medições é mostrado na Tabela 15. Para os *benchmarks* `531.deepsjeng_r` e `525.x264_r`, o tempo de execução com a heurística modificada foi menor do que o tempo de execução original, apresentando um *speedup* de 2,45% e 0,64% respectivamente. Para os outros programas, houve uma pequena piora, tal que o *benchmark* `508.namd_r` apresentou o pior resultado com um *speedup* de -1,11%.

Benchmark	**Tempo de execução (s)**		***Speedup*** **(%)**
	Original	**Heurística**	
`505.mcf_r`	**406,08**	408,50	-0,59%
`508.namd_r`	**236,97**	239,62	-1,11%
`525.x264_r`	198,17	**196,91**	0,64%
`531.deepsjeng_r*`	280,90	**274,19**	2,45%
`541.leela_r`	**427,56**	429,11	-0,36%

Tabela 15 – Comparação entre os tempos de execução dos programas compilados com a heurística original e a obtida via processo evolutivo do *benchmark* `531.deepsjeng_r`.

Como esperado, o melhor resultado foi observado no programa para o qual o treinamento foi realizado no modelo de programação genética. O comportamento dos outros casos de teste pode ser explicado pelo nível de similaridade de cada um dos *benchmarks* com o objeto de treinamento. Pode-se intuir que programas com fluxos de execução e perfis de utilização de variáveis similares aos observados em `531.deepsjeng_r` se beneficiariam da heurística modificada. Contudo, quais fatores são responsáveis pelo funcionamento exato da heurística e em quantas categorias de utilização de variáveis os programas podem ser separados ainda é desconhecido.

O treinamento com o programa `508.namd_r` resultou na heurística à direita na

Figura 47 e os resultados das validações com o restante dos *benchmarks* são mostrado na Tabela 16. Dessa vez, a heurística encontrada teve um melhor desempenho nos programas `508.namd_r` e `525.x264_r`, de 0,89% e supreendentes 1,29%. Para os programas `505.mcf_r`, `531.deepsjeng_r` e `541.leela_r` houveram pioras de -4%, -0,39% e -0,54%, respectivamente.

Dessa vez, o melhor resultado não foi observado no programa utilizado durante o treinamento. Contudo, deve-se ressaltar que a diferença entre as *workloads* de treinamento e referência para o *benchmark* `508.namd_r` é meramente o número de iterações passadas como argumento na chamada do executável. Sendo assim, a utilização de maiores números de iterações poderiam levar a um *speedup* maior.

Benchmark	**Tempo de execução (s)**		***Speedup* (%)**
	Original	**Heurística**	
`505.mcf_r`	**406,08**	422,99	-4,00%
`508.namd_r*`	236,97	**234,89**	0,89%
`525.x264_r`	198,17	**195,64**	1,29%
`531.deepsjeng_r`	**280,90**	282,01	-0,39%
`541.leela_r`	**427,56**	429,89	-0,54%

Tabela 16 – Comparação entre os tempos de execução dos programas compilados com a heurística original e a obtida via processo evolutivo do *benchmark* `508.namd_r`.

Também foi medida a quantidade de acessos à memória em tempo de execução para os cinco programas. As estatísticas para as heurística obtidas em comparação com a heurística original são mostradas na Tabela 17. É possível observar que não há uma relação direta entre a quantidade de acessos à memória e tempo de execução, como no caso do *benchmark* `531.deepsjeng_r` usando a heurística treinada com o próprio código-fonte. A latência das instruções de acesso à memória não é constante, variando a depender do tipo de dado requisitado e de sua presença em cache. Ainda assim, esses dados podem dar indicativos da influência das heurísticas no consumo energético do programa.

Benchmarks	**Original**		`531.deepsjeng_r`		`508.namd_r`	
	Reads	***Writes***	***Reads***	***Writes***	***Reads***	***Writes***
`505.mcf_r`	$4{,}420 \times 10^{11}$	$1{,}165 \times 10^{11}$	$4{,}338 \times 10^{11}$	$1{,}172 \times 10^{11}$	$4{,}682 \times 10^{11}$	$1{,}402 \times 10^{11}$
`508.namd_r`	$6{,}255 \times 10^{11}$	$1{,}486 \times 10^{11}$	$6{,}359 \times 10^{11}$	$1{,}472 \times 10^{11}$	$6{,}369 \times 10^{11}$	$1{,}494 \times 10^{11}$
`525.x264_r`	$4{,}500 \times 10^{11}$	$1{,}295 \times 10^{11}$	$4{,}526 \times 10^{11}$	$1{,}294 \times 10^{11}$	$4{,}451 \times 10^{11}$	$1{,}307 \times 10^{11}$
`531.deepsjeng_r`	$4{,}779 \times 10^{11}$	$2{,}281 \times 10^{11}$	$4{,}805 \times 10^{11}$	$2{,}291 \times 10^{11}$	$4{,}811 \times 10^{11}$	$2{,}303 \times 10^{11}$
`541.leela_r`	$6{,}036 \times 10^{11}$	$2{,}408 \times 10^{11}$	$6{,}037 \times 10^{11}$	$2{,}406 \times 10^{11}$	$6{,}023 \times 10^{11}$	$2{,}403 \times 10^{11}$

Tabela 17 – Quantidades de acessos à memória em tempo de execução para a heurística original e as duas heurísticas obtidas através do algoritmo evolutivo.

8 CONCLUSÃO

A alocação de registradores é, sem dúvida, uma das otimizações de código mais importantes do processo de compilação. O acesso de valores em memória é responsável por uma parcela significativa do tempo de execução e consumo energético de programas compilados e, por isso, não deve-se poupar esforços para que sejam desenvolvidos algoritmos e técnicas de minimização de *spill code* cada vez melhores. Nesse cenário, as técnicas de inteligência artificial, como aprendizado de máquina e a computação evolutiva, podem fornecer grande ajuda, dada a natureza do problema e a indisponibilidade de soluções exatas.

Neste trabalho foi feita uma revisão abrangente sobre o estado da arte das técnicas de alocação de registradores, minimização de *spill code* e inteligência artificial, a fim de estudar as possibilidades de combinação entre essas várias técnicas. Foram contemplados diversos trabalhos clássicos e recentes nessas áreas, sendo possível vislumbrar a vasta gama de oportunidades para a utilização da inteligência artificial na alocação de registradores, tanto na melhoria de técnicas tradicionais como em abordagens totalmente novas.

Apoiada na revisão da literatura feita anteriormente, uma proposta experimental baseada em algoritmos evolutivos, área da inteligência artificial, para minimização de *spill code* foi idealizada. O modelo foi implementado e foram realizados experimentos para avaliar a competência das técnicas usadas, e foram obtidos resultados animadores. Isso não só demonstra a viabilidade deste tipo de implementação, bem como serve de perspectiva para novos trabalhos que sigam na mesma linha.

Sendo assim, seguem como pontos em aberto e potenciais temas de investigação para trabalhos futuros, tarefas tais quais:

- Explorar outras abordagens de algoritmo evolutivo para o treinamento de heurísticas;
- Desvendar a relação entre as características do programa e o desempenho da heurística de alocação;
- Avaliar a integração de técnicas de *machine learning*, possivelmente usada em conjunto com algoritmos evolutivos, na alocação de registradores;
- Obtenção de *datasets* sobre programas complexos, que sirvam como referência para os métodos de inteligência artificial.

REFERÊNCIAS

[1] CHAITIN, G. J. Register Allocation & Spilling via Graph Coloring. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. [S.l.]: Association for Computing Machinery, 1982. (SIGPLAN '82), p. 98–105.

[2] POLETTO, M.; SARKAR, V. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, v. 21, n. 5, p. 895–913, 1999.

[3] WIMMER, C. *Linear scan register allocation for the Java HotSpotTM client compiler*. Dissertação (Mestrado) — Johannes Kepler University, 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, Alemanha, 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, v. 32, n. 5, p. 287–295, 1997.

[6] COOPER, K. D.; SIMPSON, L. T. Live range splitting in a graph coloring register allocator. In: KOSKIMIES, K. (Ed.). *Compiler Construction*. [S.l.]: Springer, 1998. p. 174–187.

[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, 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).

[10] CORPUS, K. R. M. et al. Coupling covariance matrix adaptation with continuum modeling for determination of kinetic parameters associated with electrochemical CO2 reduction. *Joule*, v. 7, n. 6, p. 1289–1307, 2023.

[11] EIBEN, A. E.; SMITH, J. E. *Introduction to Evolutionary Computing*. 2nd. ed. [S.l.]: Springer, Incorporated, 2015.

[12] 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*. [S.l.]: Association for Computing Machinery, 2003. (PLDI '03), p. 77–90.

[13] YU, Y. et al. A review of recurrent neural networks: LSTM cells and network architectures. *Neural Computation*, MIT Press, v. 31, n. 7, p. 1235–1270, 2019.

[14] 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.

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

[16] PUTERMAN, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1. ed. [S.l.]: John Wiley & Sons, Inc., 1994.

[17] MOHAMMED, M.; KHAN, M.; BASHIER, E. *Machine Learning: Algorithms and Applications*. 1. ed. [S.l.]: CRC Press, 2016.

[18] ASHLOCK, D. *Evolutionary Computation for Modeling and Optimization*. 1. ed. [S.l.]: Springer, 2006.

[19] SPEC CPU® 2017 Overview. 2018. <https://www.spec.org/cpu2017/Docs/overview.html>. Acesso em: 16 de junho de 2023.

[20] AHO, A. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. [S.l.]: Addison-Wesley, 2007. (Alternative eText Formats Series).

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

[22] MITTAL, S. A Survey of Techniques for Designing and Managing CPU Register File. *Concurrency and Computation Practice and Experience*, v. 29, 2016.

[23] BRIGGS, P. *Register Allocation via Graph Coloring*. Tese (Doutorado) — Rice University, Houston, EUA, 1992.

[24] 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.

[25] PEREIRA, F. M. Quintão; PALSBERG, J. Register allocation by puzzle solving. *SIGPLAN Not.*, Association for Computing Machinery, v. 43, n. 6, p. 216–226, 2008.

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

[27] KARP, R. M. *Reducibility among Combinatorial Problems*. [S.l.]: Springer, 1972. 85-103 p.

[28] GAREY, M. R.; JOHNSON, D. S. The complexity of near-optimal graph coloring. *J. ACM*, Association for Computing Machinery, v. 23, n. 1, p. 43–49, 1976.

[29] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. *SIGPLAN Not.*, Association for Computing Machinery, v. 24, n. 7, p. 258–263, 1989.

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

[31] 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*. [S.l.]: Association for Computing Machinery, 1998. (PLDI '98), p. 142–151.

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

[33] RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: a Modern Approach*. 4th. ed. [S.l.]: Pearson, 2021.

[34] 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. 1st International Conference on Advances in Information, Computing and Trends in Data Engineering (AICDE - 2020).

[35] BäCK, T.; FOGEL, T.; MICHALEWICZ, D. *Evolutionary Computation 1: Basic Algorithms and Operators*. [S.l.]: Institute of Physics Publishing, Bristol and Philadelphia, 2000.

[36] The LLVM Compiler Infrastructure. 2001. Acesso em: 16 de junho de 2023. Disponível em: <https://llvm.org>.

[37] ALLEN, F. E. Control flow analysis. In: *Proceedings of a Symposium on Compiler Optimization*. [S.l.]: Association for Computing Machinery, 1970. p. 1–19.

[38] AMD64 ® architecture programmer's manual volume 2: System programming. 2024. <https://www.amd.com/en/search/documentation/hub.html>. Acesso em: 22 de abril de 2024.

[39] 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.

[40] GEORGE, L.; APPEL, A. W. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, v. 18, n. 3, p. 300–324, 1996.

[41] BRAUN, M.; HACK, S. Register Spilling and Live-Range Splitting for SSA-Form Programs. In: MOOR, O. de; SCHWARTZBACH, M. I. (Ed.). *Compiler Construction*. [S.l.]: Springer, 2009. p. 174–189.

[42] CHAITIN, G. J. et al. Register Allocation via Coloring. *Computer Languages*, v. 6, n. 1, p. 47–57, 1981.

[43] 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*. [S.l.]: Association for Computing Machinery, 2016. (PPPJ '16).

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

[45] BJöRKLUND, A.; HUSFELDT, T.; KOIVISTO, M. Set Partitioning via Inclusion-Exclusion. *SIAM Journal on Computing*, v. 39, n. 2, p. 546–563, 2009.

[46] BRIGGS, P. et al. Coloring Heuristics for Register Allocation. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. [S.l.]: Association for Computing Machinery, 1989. (PLDI '89), p. 275–284.

[47] 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.

[48] SCHOLZ, B.; ECKSTEIN, E. Register Allocation for Irregular Architectures. *SIGPLAN Not.*, Association for Computing Machinery, v. 37, n. 7, p. 139–148, 2002.

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

[50] BUCHWALD, S.; ZWINKAU, A. Instruction selection by graph transformation. In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. [S.l.]: Association for Computing Machinery, 2010. (CASES '10), p. 31–40.

[51] 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.

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

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

[54] 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.

[55] WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Association for Computing Machinery, v. 13, n. 2, p. 181–210, 1991.

[56] CALLAHAN, D.; KOBLENZ, B. Register allocation via hierarchical graph coloring. *ACM Sigplan Notices*, Association for Computing Machinery, v. 26, n. 6, p. 192–203, 1991.

[57] USAMA, M. et al. Unsupervised machine learning for networking: Techniques, applications and research challenges. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc. (IEEE), v. 7, p. 65579 – 65615, 2019.

[58] PRAKASH, V. J.; NITHYA, L. M. A survey on semi-supervised learning techniques. *International Journal of Computer Trends and Technology (IJCTT)*, v. 8, n. 1, p. 25–29, 2014.

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

[60] KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, AI Access Foundation, v. 4, n. 1, p. 237–285, 1996.

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

[62] MATHEW, A.; AMUDHA, P.; SIVAKUMARI, S. Deep learning techniques: An overview. In: *Advanced Machine Learning Technologies and Applications*. [S.l.]: Springer, 2021. p. 599–608.

[63] 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.

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

[65] THIERENS, D.; GOLDBERG, D. E. Mixing in genetic algorithms. In: *Proceedings of the 5th International Conference on Genetic Algorithms*. [S.l.]: Morgan Kaufmann Publishers Inc., 1993. p. 38–47.

[66] HOLLAND, J. H. Outline for a logical theory of adaptive systems. *J. ACM*, Association for Computing Machinery, v. 9, n. 3, 1962.

[67] BREMERMANN, H. J. et al. Optimization through evolution and recombination. *Self-organizing systems*, Washington, DC: Spartan, v. 93, p. 106, 1962.

[68] FRASER, A. Simulation of Genetic Systems by Automatic Digital Computers. *Australian Journal of Biological Sciences*, 1957.

[69] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. [S.l.]: The MIT Press, 1992.

[70] VIKHAR, P. A. Evolutionary algorithms: A critical review and its future prospects. In: IEEE. *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. [S.l.], 2016. p. 261–265.

[71] CRAMER, N. L. A Representation for the Adaptive Generation of Simple Sequential Programs. In: *Proceedings of the 1st International Conference on Genetic Algorithms*. [S.l.]: L. Erlbaum Associates Inc., 1985. p. 183–187.

[72] KOZA, J. R. Hierarchical genetic algorithms operating on populations of computer programs. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*. [S.l.]: Morgan Kaufmann Publishers Inc., 1989. (IJCAI'89), p. 768–774.

[73] KOZA, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, v. 4, n. 2, p. 87–112, 1994.

[74] RECHENBERG, I. *Cybernetic Solution Path of an Experimental Problem*. [S.l.]: Royal Aircraft Establishment, Ministry of Aviation, Great Britain, 1965. (RAE-LT-1122).

[75] SCHWEFEL, H.-P.; RUDOLPH, G. Contemporary evolution strategies. In: MORÁN, F. et al. (Ed.). *Advances in Artificial Life*. [S.l.]: Springer, 1995. p. 891–907.

[76] FOGEL, L. J.; OWENS, A. J.; WALSH, M. J. Intelligent decision making through a simulation of evolution. *Behavioral Science*, v. 11, n. 4, p. 253–272, 1966.

[77] CHEN, X. et al. A multi-facet survey on memetic computation. *Trans. Evol. Comp*, IEEE, v. 15, n. 5, p. 591–607, 2011.

[78] STEPHENSON, M. W. *Automating the construction of compiler heuristics using machine learning*. Tese (Doutorado) — Massachusetts Institute of Technology, 2006. AAI0810106.

[79] KRI, F.; FEELEY, M. Genetic instruction scheduling and register allocation. In: *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile*. [S.l.]: IEEE, 2004. p. 76–83.

[80] MAHAJAN, A.; ALI, M. S. Hybrid evolutionary algorithm for graph coloring register allocation. In: IEEE. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. [S.l.], 2008. p. 1162–1167.

[81] TOPCUOGLU, H. R.; DEMIROZ, B.; KANDEMIR, M. Solving the Register Allocation Problem for Embedded Systems using a Hybrid Evolutionary Algorithm. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 11, n. 5, p. 620–634, 2007.

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

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

[84] 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.

[85] 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.

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

[87] 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.

[88] DODARO, C. et al. Deep learning for the generation of heuristics in answer set programming: A case study of graph coloring. In: *Logic Programming and Nonmonotonic Reasoning*. [S.l.]: Springer, 2022. p. 145–158.

[89] MUSLIU, N.; SCHWENGERER, M. Algorithm selection for the graph coloring problem. In: *Learning and Intelligent Optimization*. [S.l.]: Springer, 2013. p. 389–403.

[90] LATTNER, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. Dissertação (Mestrado) — Computer Science Dept., University of Illinois Urbana-Champaign, Champaign, EUA, 2002.

[91] FORTIN, F.-A. et al. DEAP: evolutionary algorithms made easy. *The Journal of Machine Learning Research*, JMLR.org, v. 13, n. 1, p. 2171–2175, 2012.

[92] XAVIER, T. C. d. S. et al. A Detailed Analysis of the LLVM's Register Allocators. In: IEEE. *2012 31st International Conference of the Chilean Computer Science Society*. [S.l.], 2012. p. 190–198.

[93] BACH, M. et al. Analyzing Parallel Programs with PIN. *Computer*, v. 43, n. 3, p. 34–41, 2010.