# Alocação de Registradores Utilizando Coloração em Grafos

Raoni Silva, Raul Ramalho

13 de julho de 2025

Turma de Grafos 35M34
Unidade 3

# Sumário

# 1 Resumo

Investigação do estado da arte sobre otimização do uso de registradores com variáveis pelas técnicas de modelagem com coloração de grafos de interferência. Investigação de sua evolução histórica, mensuração dos resultados e desfecho.

# 2 Introdução

Registradores são componentes de memória de uso genérico de uma CPU de um computador. Sua função é de armazenar dados e intruções que serão processados de imediato. É a tecnologia de maior nível na hierárquia de memória em um computador.

Pela sua quantidade reduzida, uso intermitente e alta importância, sua ociosidade ou desperdício não é desejável. O uso ótimo destes componentes é portante prioritário. Para isso, pesquisa e técnicas de otimização foram desenvolvidas nesta área.

Entre as diversas abordagens disponíveis, uma delas adotadas fora a de coloração de grafos de interferência. Técnica que consiste em cada variável em dado programa de computador é representada por um vértice no grafo, e suas arestas representam a coexistência do tempo de vida desta variável no mesmo instante. Cada cor atribuída ao grafo corresponde ao número de registradores de propósito geral disponíveis.

Inicialmente, em 1981 nos laboratórios da IBM, o cientista Chaitin e seus colegas de pesquisa desenvolveram o primeiro algoritmo otimização do uso de registradores com coloração de grafos de interferência.

O algoritmo é especificado em quatro etapas, construção do grafo de interferência, armazenamento na memória e coloração, spilling ( armazenamento na ram) e recuperação das variáveis na ram e recosntrução do grafo.

Posteriormente, o cientista Briggs (1992) e seus colegas da Rice University, interessados no problema decidem refinar o algoritmo de Chaitin (1982) e companhia, resultando na aprimoração das heurísticas na etapa de coalescência, simplificação e spilling(derramamento).

Por fim, em 1996, pelo Bell Labs na equipe de George e Appel et al., novamente, a técnica de coloração de grafos de interferência fora otimizada mais uma vez apartir de Briggs e companhia. Agora, a técnica de coalescência admite uma nova heurística, critérios mais robustos foram atribuidas como prevenção do pior caso do algoritmo de seus antecessores. Uma nova etapa de congelamento fora adiciona antes de ocorrer o spilling. Atingindo, assim, o estado da arte.

# 3 Descrição do Problema

O problema de alocação de registradores consiste em atribuir variáveis a um número limitado de registradores durante a compilação de um programa.

## 3.1 Objetivo do problema

O objetivo é estabelecer uma associação entre as variáveis do programa e os registradores disponíveis, determinando, sempre que possível, a qual registrador cada variável será atribuída.

Uma solução considerada ótima é aquela que utiliza o menor número possível de registradores para alocar todas as variáveis do programa.

Por exemplo, considere um programa com cinco variáveis. Em uma primeira tentativa de alocação, cada variável é atribuída a um registrador diferente, totalizando cinco registradores utilizados. Essa é uma solução válida. No entanto, se for possível alocar todas as cinco variáveis utilizando apenas três registradores, essa será uma solução melhor. Além disso, se o número disponível de registradores for de fato três, essa alocação representa uma solução ótima, pois atende à limitação de recursos.

Em certos casos, pode não existir uma solução válida com o número disponível de registradores, ou seja, não é possível atribuir cada variável a um registrador sem conflitos. Nesses cenários, a literatura propõe abordagens alternativas, como a técnica de *spilling*, que transfere algumas variáveis da memória rápida (registradores) para a memória principal (RAM) durante a execução.

## 3.2 Entradas esperadas

A entrada do problema consiste em uma descrição das variáveis do programa e suas interações, indicando quais variáveis estão ativas ao mesmo tempo e, portanto, não podem compartilhar o mesmo registrador.

Essa descrição pode assumir diferentes formas, como uma tabela de intervalos de uso de variáveis (lifetimes), uma matriz de interferência, ou qualquer outro formato que permita inferir conflitos de uso simultâneo entre variáveis.

## 3.3 Saídas esperadas

A saída deve indicar se existe uma atribuição válida das variáveis aos registradores disponíveis. Caso exista, a solução deve apresentar a alocação realizada, especificando a qual registrador cada variável foi atribuída.

Em caso de impossibilidade (isto é, se não for possível realizar a alocação com o número de registradores fornecido), a saída deve indicar que não há solução válida. Alternativamente, pode apresentar uma solução com *spilling*, identificando quais variáveis foram movidas para a memória.

# 4 Modelagem em grafos

Para a modelagem do problema, considere os vértices como representações das variáveis do programa, e as arestas indicam conflitos entre variáveis — isto é, situações em que duas variáveis estão ativas ao mesmo tempo e, portanto, não podem compartilhar o mesmo registrador. O grafo formado por essa relação é chamado de *grafo de interferência*.

Dessa forma, uma coloração adequada do grafo de interferência — atribuindo uma cor diferente para cada conjunto de variáveis que não podem ser simultâneas — corresponde a uma alocação de registradores sem conflitos.

Se a coloração do grafo puder ser feita com um número de cores menor ou igual ao número de registradores disponíveis, então é possível realizar a alocação diretamente. Caso contrário, é necessário utilizar alguma técnica de spilling para lidar com as variáveis que não puderem ser atribuidas a um registrador.

# 5 Estado da Arte

o estado da arte foi alcançado pelo laboratório bell labs innovations através do algoritmo de George-Appel (1996). o algoritmo consiste no aprimoramento dos trabalho de Briggs (1992) e Chaitin (1982).

a diferença mais aguda entre o trabalho de briggs e george e appel foi a introdução de uma etapa de freezing ( congelamento ) antes de spillar ( derramar ) a variável na cpu. esse estado intermediário decidirá quando a variável deve ser reintroduzida no grafo original ou se será armazenada à ram.

também fora incrementado a heurística de coalescência, tornando-o mais agressiva com a regras de briggs permitindo uma maior eficácia na fusão de variáveis.

# 6 Descrição do Algoritmo

O algoritmo de George-Appel funciona através da construção de um grafo de interferência iterativo, e posteriormente, K-colorido, onde K é o número de registradores. Ao todo, possui sete etapas. *build, simplify, conservative coalesce, freeze, potential spill, select e actual spill.* Adota o estilo de Coalescência Conservadora de Briggs e a simplificação de Chaitin resultando em uma maior eficiência que ambas as técnicas separadas. A nova abordagem para alocação de registradores com grafos de interferência iterativo garante uma execução em loop das etapas, retomando, em laço, as etapas de simplficação, coalescência, congelamento e derramamento inúmeras vezes no mesmo grafo. Tal característica torna dificil estimar a complexidade do algoritmo.

*Build:* Constrói o grafo de interferência e categoriza os vértices entre instruções de cópia (move related) ou não.

*Simplify:* Remove as instruções de cópia de menor grau no grafo.

*Conservative Coalesce:* Atua semelhante ao algoritmo de Briggs.

*Freeze:* Nem simplifica nem coalesce, busca instruções cópia de baixo grau e as congela. Retorna às etapas de simplificação e coalescência

*Select:* Mesma etapa dos algoritmos anteriores, exceto que não adotada bias na hora do julgamento para otimização precoce de instrução de moves.

*Spill:* Armazena as variáveis na memória RAM para retorná-las futuramente.

Uma instrução de cópia ( ou move instruction ) é quando uma variável está armazenada no registrador temporário de origem e passa ao registrador de destino ( onde será processada ).

## 6.1 Pseudocódigo e Complexidade:

Dado os retornos sucessivos à etapa de build, e as sucessivas iterações e interoperações entre etapas de congelamento, derramento, coalescência, simplificação e seleção. É de enorme dificuldade estimar a complexidade global do algoritmo.

O pseudocódigo utilizado fora o original do artigo de George-Appel et al, 1996, página 19 à 23.

```
procedure Main()
    LivenessAnalysis()
    Build()
    MkWorklist()
    repeat
        if simplifyWorklist ≠ {} then Simplify()
        else if worklistMoves ≠ {} then Coalesce()
        else if freezeWorklist ≠ {}  then Freeze()
        else if spillWorklist ≠ {} then SelectSpill()
    until simplifyWorklist = {}  ∧ worklistMoves = {}
         ∧ freezeWorklist = {}  ∧ spillWorklist = {}
    AssignColors()
    if spilledNodes ≠ {} then
        RewriteProgram(spilledNodes)
        Main()
```

```
procedure AddEdge(u, v)
    if ((u,v) ∉ adjSet) ∧ (u <> v) then
        adjSet := adjSet ∪ {(u,v),(v,u)}
        if u ∉ precolored then
            adjList[u] := adjList[u] ∪ {v}
            degree[u] := degree[u] + 1
        if v ∉ precolored then
            adjList[v] := adjList[v] ∪ {u}
            degree[v] := degree[v] + 1

procedure Build ()
    forall b ∈ blocks in program
        let live = liveOut(b)
        forall I ∈ instructions(b) in reverse order
            if isMoveInstruction(I) then
                live := live\use(I)
                forall n ∈ def(I) ∪ use(I)
                    moveList[n] := moveList[n] ∪ {I}
                worklistMoves := worklistMoves ∪ {I}
            live := live ∪ def(I)
            forall d ∈ def(I)
                forall l ∈ live
                    AddEdge(l, d)
            live := use(I) ∪ (live\def(I))

        function Adjacent(n)
            adjList[n] \ (selectStack ∪ coalescedNodes)

        function NodeMoves (n)
            moveList[n] ∩ (activeMoves ∪ worklistMoves)

        function MoveRelated(n)
            NodeMoves(n) ≠ {}

        procedure MkWorklist()
            forall n ∈ initial
                initial := initial \ {n}
                if degree[n] ≥ K then
                    spillWorklist := spillWorklist ∪ {n}
                else if MoveRelated(n) then
                    freezeWorklist := freezeWorklist ∪ {n}
                else
                    simplifyWorklist := simplifyWorklist ∪ {n}

        procedure Simplify()
            let n ∈ simplifyWorklist
            simplifyWorklist := simplifyWorklist \ {n}
            push(n, selectStack)
            forall m ∈ Adjacent(n)
                DecrementDegree(m)
```

```
    procedure DecrementDegree(m)
        let d = degree[m]
        degree[m] := d-1
        if d = K then
            EnableMoves({m} ∪ Adjacent(m))
            spillWorklist := spillWorklist \ {m}
            if MoveRelated(m) then
                freezeWorklist := freezeWorklist ∪ {m}
            else
                simplifyWorklist := simplifyWorklist ∪ {m}

    procedure EnableMoves(nodes)
        forall n ∈ nodes
            forall m ∈ NodeMoves(n)
                if m ∈ activeMoves then
                    activeMoves := activeMoves \ {m}
                    worklistMoves := worklistMoves ∪ {m}
```

*Coalesce*

```
procedure Coalesce()
    let m(=copy(x,y)) ∈ worklistMoves
    x := GetAlias(x)
    y := GetAlias(y)
    if y ∈ precolored then
        let (u, v) = (y, x)
    else
        let (u, v) = (x, y)
    worklistMoves := worklistMoves \ {m}
    if (u = v) then
        coalescedMoves := coalescedMoves ∪ {m}
        AddWorkList(u)
    else if v ∈ precolored ∨ (u, v) ∈ adjSet then
        constrainedMoves := constrainedMoves ∪ {m}
        addWorkList(u)
        addWorkList(v)
    else if u ∈ precolored ∧ (∀t ∈ Adjacent(v), OK(t, u))
          ∨ u ∉ precolored ∧ Conservative(Adjacent(u) ∪ Adjacent(v)) then
        coalescedMoves := coalescedMoves ∪ {m}
        Combine(u,v)
        AddWorkList(u)
    else
        activeMoves := activeMoves ∪ {m}
```

*AddWorkList*

```
procedure AddWorkList(u)
    if (u ∉ precolored  ∧ not(MoveRelated(u))  ∧ degree[u] < K) then
        freezeWorklist := freezeWorklist \ {u}
        simplifyWorklist := simplifyWorklist ∪ {u}
```

*OK*

```
function OK(t,r)
    degree[t] < K ∨ t ∈ precolored ∨ (t, r) ∈ adjSet
```

*Conservative*

```
function Conservative(nodes)
    let k = 0
    forall n ∈ nodes
        if degree[n] ≥ K then k := k + 1
    return (k < K)
```

*GetAlias*

```
function GetAlias (n)
    if n ∈ coalescedNodes then
        GetAlias(alias[n])
    else n
```

```
procedure Combine(u,v)
    if v ∈ freezeWorklist then
        freezeWorklist := freezeWorklist \ {v}
    else
        spillWorklist := spillWorklist \ {v}
    coalescedNodes := coalescedNodes ∪ {v}
    alias[v] := u
    nodeMoves[u] := nodeMoves[u] ∪ nodeMoves[v]
    forall t ∈ Adjacent(v)
        AddEdge(t,u)
        DecrementDegree(t)
    if degree[u] ≥ K ∧ u ∈ freezeWorkList
        freezeWorkList := freezeWorkList \ {u}
        spillWorkList := spillWorkList ∪ {u}

procedure Freeze()
    let u ∈ freezeWorklist
    freezeWorklist := freezeWorklist \ {u}
    simplifyWorklist := simplifyWorklist ∪ {u}
    FreezeMoves(u)
```

```
procedure FreezeMoves(u)
    forall m(= copy(u,v) or copy(v,u)) ∈ NodeMoves(u)
        if m ∈ activeMoves then
            activeMoves := activeMoves \ {m}
        else
            worklistMoves := worklistMoves \ {m}
        frozenMoves := frozenMoves ∪ {m}
        if NodeMoves(v) = {} ∧ degree[v] < K then
            freezeWorklist := freezeWorklist \ {v}
            simplifyWorklist := simplifyWorklist ∪ {v}

procedure SelectSpill()
    let m ∈ spillWorklist selected using favorite heuristic
        Note: avoid choosing nodes that are the tiny live ranges
        resulting from the fetches of previously spilled registers
    spillWorklist := spillWorklist \ {m}
    simplifyWorklist := simplifyWorklist ∪ {m}
    FreezeMoves(m)

procedure AssignColors()
    while SelectStack not empty
        let n = pop(SelectStack)
        okColors := {0, . . . , K-1}
        forall w ∈ adjList[n]
            if GetAlias(w) ∈ (coloredNodes ∪ precolored) then
                okColors := okColors \ {color[GetAlias(w)]}

        if okColors = {} then
            spilledNodes := spilledNodes ∪ {n}
        else
            coloredNodes := coloredNodes ∪ {n}
            let c ∈ okColors
            color[n] := c
    forall n ∈ coalescedNodes
        color[n] := color[GetAlias(n)]

procedure RewriteProgram()
    Allocate memory locations for each v ∈ spilledNodes,
    Create a new temporary v_i for each definition and each use,
    In the program (instructions), insert a store after each
    definition of a v_i, a fetch before each use of a v_i.
    Put all the v_i into a set newTemps.
    spilledNodes := {}
    initial := coloredNodes ∪ coalescedNodes ∪ newTemps
    coloredNodes := {}
    coalescedNodes := {}
```

Assume-se, que a instânciação do pseudocódigo ao algoritmo em uma linguagem de programação qualquer o implementador usará estruturas de dados eficientes para tal.

Seja A o número de arestas de interferência, V o número de vértices, K o número de registradores e M o número de instruções.

*addEdge:* Abrevia-se a complexidade dessa instrução para $O(1)$.

*Build:* Rotina responsável por criar os vértices e arestas, logo $O(V + A)$.

*MkWorklist:* Itera até $n$ vezes o tamanho de *initial*, logo $O(n)$.

*Simplify:* Dado um nó $n$, itera sobre seus vizinhos $adj(n)$, o que custa $O(d(n))$.

*DecrementDegree:* Poucas comparações, justificamente análoga à addEdge.

*EnableMoves:* Itera sobre os nós e a lista de instruções. Custo total $O(V+M)$.

*Coalesce:* O custo de coalesce depende da complexidade dos vértices vizinhos chamados para a coalescência, logo $O(adj(u) + adj(v))$.

*Combine:* Itera sobre todos os vertices pertencentes à adjacente, logo $O(d(vertice))$.

*Freeze:* Custo $O(1)$ pois evoca FreezeMoves.

*FreezeMoves:* Custo $O(M)$ pois itera o vetor do número de instruções.

*SelectSpill:* Depende diretamente de quanto derramamento será causado no programa, pode ser $O(1)$ ou $O(V)$.

*AssignColors:* Irá atribuir cores a todos vértices, logo $O(V)$.

A soma das complexidades individuais não representa a complexidade global do algoritmo, dado que o processo iterativo ocorre inúmeras vezes e chamadas de congelamento, derramamento, simplificação ocorrem inúmeras vezes no mesmo código.

# 7 Descrição dos Experimentos Computacionais

## 7.1 Metodologia de testes

Os experimentos foram feitos a partir de um gerador customizado de instâncias de testes, onde é possível simular programas da forma mais fiel possível. Foram gerados 15 instâncias de programas com características variadas, e o algoritmo de George-Appel foi executado 3 vezes para cada instância. Por meio dessa base de dados é possível obter métricas sobre os resultados obtidos, tais como:

**Tempo de execução:** Medido em milissegundos, representa o tempo de cada execução individual da heuristica com uma certa entrada de dados.

**Quantidade de variáveis:** Representa a quantidade total de variáveis ao final do processo de iterativamente simplificar, coaslescer os nós e reescrever o código.

**Quantidade de variáveis despejadas:** Ao fim da quarta execução, na nossa implementação o algoritmo desiste de encontrar uma solução melhor que reduza a quantidade de nós despejados e finaliza sua execução.

### 7.1.1 Simulador de programas

Para conseguir entradas válidas para o algoritmo, é necessário prover de antemão quais são as variáveis, em que linhas as variáveis são utilizadas e também, quais são os nós provenientes de operações de moves. Para isso, foi construido um simulador simplificado de programas, ele recebe como entrada parâmetros tais como:

**ID:** Um identificador para salvar em um arquivo os dados simulados.

**Número de variáveis:** Quantidade de variáveis que serão simuladas no programa.

**Quantidade de moves:** Quantidade de arestas que serão criadas como operações de moves, essas arestas são especiais pois, na prática, significa que o início da vida de uma das variáveis é exatamente no final da vida da outra variável, em uma operação do tipo (x = y). Dessa forma, essa informação deve ser considerada ao gerar a lista de linhas de usos de cada variável.

**Limite inferior:** Menor quantidade de linhas de código em que a variável é utilizada.

**Limite superior:** Maior quantidade de linhas de código em que a variável é utilizada.

**Linhas de código:** Quantidade de linhas de código totais no código.

Com isso, foram gerados casos de teste variando principalmente o número de variáveis: 10, 100, 200, 500 e 1000. E o limite superior de usos de variável: 5, 15, 30. Com cada combinação desses valores sendo uma instância de entrada de dados para o algoritmo. Além disso, foram alteradas a quantidade de moves,

mas os resultados não melhoraram ou pioraram, dessa forma, os experimentos não foram incluídos no teste. Como padrão, foi utilizado 32 como a quantidade de registradores disponíveis para coloração.

### 7.1.2 Plataforma de execução

Todas as execuções foram realizadas em uma única máquina para melhorar a consistência dos dados obtidos. A configuração do ambiente foi:

- Hardware:

  **CPU:** Ryzen 5 5600

  **RAM:** 16GB DDR4 2400Mhz

- Software:

  **Sistema Operacional:** Fedora 41

  **Linguagem:** Python 3.13

  **Project Manager:** UV(astral-sh (s.d.))

## 7.2 Como reproduzir os testes

Para a reprodução dos experimentos, recomenda-se o uso da ferramenta de gerenciamento de pacotes **UV** (astral-sh (s.d.)) para a criação de um ambiente Python consistente.

O processo consiste em dois passos principais:

1. **Gerar as instâncias de entrada:** Execute o comando abaixo no terminal para criar os cinco arquivos de teste que servirão de entrada para o algoritmo.

   ```
   uv run src/generator.py
   ```

2. **Executar o algoritmo:** Após a geração das instâncias, utilize o seguinte comando para executar o algoritmo de coloração de grafos sobre os cinco arquivos gerados.

   ```
   uv run src/george-appel.py
   ```

# 8 Resultados Obtidos

A execução dos experimentos em um conjunto de instâncias variadas permitiu uma análise aprofundada do comportamento da heurística de alocação de registradores. Os testes foram projetados para avaliar o desempenho do algoritmo sob diferentes cenários de pressão de quantidade de usos de cada variável (variando o limite superior) e a quantidade de variáveis. A seguir, são apresentados e discutidos os resultados quantitativos.

## 8.1 Resultados Quantitativos

Os experimentos foram executados três vezes para cada configuração, e os tempos de execução foram registrados. A Tabela 1 consolida estes dados, apresentando o tempo médio de execução e o desvio padrão correspondente para cada classe de teste e tamanho de instância (número de variáveis).

Tabela 1: Tempo médio de execução e desvio padrão para diferentes cenários de teste.

| **Tipo de Teste** | **Instância** | **Tempo Médio (s)** | **Desvio Padrão (s)** |
|---|---|---|---|
| teste-lim-sup-5 | 10 | 0.000135 | 0.000 006 |
| | 100 | 0.022424 | 0.000 096 |
| | 200 | 0.095923 | 0.007 000 |
| | 500 | 0.654350 | 0.004 085 |
| | 1000 | 3.753158 | 0.081 135 |
| teste-padrao | 10 | 0.000135 | 0.000 005 |
| | 100 | 0.095402 | 0.001 427 |
| | 200 | 0.573312 | 0.002 754 |
| | 500 | 3.807789 | 0.014 870 |
| | 1000 | 21.030808 | 0.207 316 |
| teste-lim-sup-30 | 10 | 0.000166 | 0.000 043 |
| | 100 | 0.328989 | 0.011 429 |
| | 200 | 1.377198 | 0.029 020 |
| | 500 | 17.993473 | 0.191 330 |
| | 1000 | 51.772303 | 0.286 317 |

## 8.2 Análise dos Resultados

A análise dos dados apresentados na Tabela 1 revela tendências importantes sobre o comportamento do algoritmo.

**Análise de Escalabilidade** Para todos os tipos de teste, observa-se um crescimento consistente no tempo de execução à medida que o número de variáveis da instância aumenta. Este crescimento é notavelmente não linear, indicando que a

complexidade do problema de coloração aumenta significativamente com o tamanho do grafo de interferência. Mesmo com a heurística tendo uma complexidade polinomial, o maior impacto no tempo de execução se dá por causa da reescrita do "programa". Pois isso adiciona uma quantidade imensa de nós para cada reescrita. Para fins de comparação, observou-se um aumento na quantidade de nós em alguma execuções com inicialmente 1000 variáveis, terminando com cerca de 8000 variáveis, devido ao split das variáveis que seriam despejadas.

**Impacto da quantidade de usos de variáveis (Limite Superior)** A variação no limite superior de usos de variáveis apresentou o impacto mais significativo no desempenho.

- **Cenário Intenso (Limite=30):** O teste `teste-lim-sup-30` foi, de longe, o que apresentou os maiores tempos de execução. Com um grande número de usos de cada variável, a liveness analysis gera intervalos cada vez maiores e com cada vez mais interferências. Além disso, quando as variáveis são divididas em variáveis com tempos de vida menor, a quantidade de variáveis aumenta muito, de forma que a próxima iteração do algoritmo com as novas variáveis demore ainda mais.

- **Cenário Restrito (Limite=5):** Inversamente, o teste `teste-lim-sup-5` foi o mais rápido. Como existiam menos usos de cada variável, duas coisas importantes acontecem, o liveness dela tende a ser menor, e principalmente, quando ela é dividida em outras variáveis temporárias, a quantidade de variáveis para a próxima execução do algoritmo não aumenta tanto assim, melhorando a eficiência do algoritmo.

- **Cenário Padrão (Limite=15):** O teste `teste-padrao`, que representa nosso caso base, situa-se entre os dois extremos, indicando um equilíbrio entre as quantidades de variáveis sendo divididas para as execuções seguintes.

## 8.3 Visualização Gráfica

Para melhor ilustrar as tendências de escalabilidade e o impacto dos parâmetros, o Gráfico 1 plota o tempo médio de execução em função do número de variáveis da instância para cada tipo de teste. A escala do eixo Y é logarítmica para acomodar a grande variação nos tempos de execução.

O gráfico evidencia claramente o comportamento discutido: a linha correspondente a `Limite=30` (vermelha) se destaca com os maiores tempos, enquanto a linha para `Limite=5` (verde) permanece consistentemente como a mais rápida para instâncias maiores. As demais configurações apresentam um desempenho intermediário e muito próximo entre si.

## 8.4 Discussão sobre a Validade das Instâncias

Uma consideração fundamental para a interpretação dos resultados é a natureza das instâncias geradas pelo simulador. Especificamente, a implementação atual

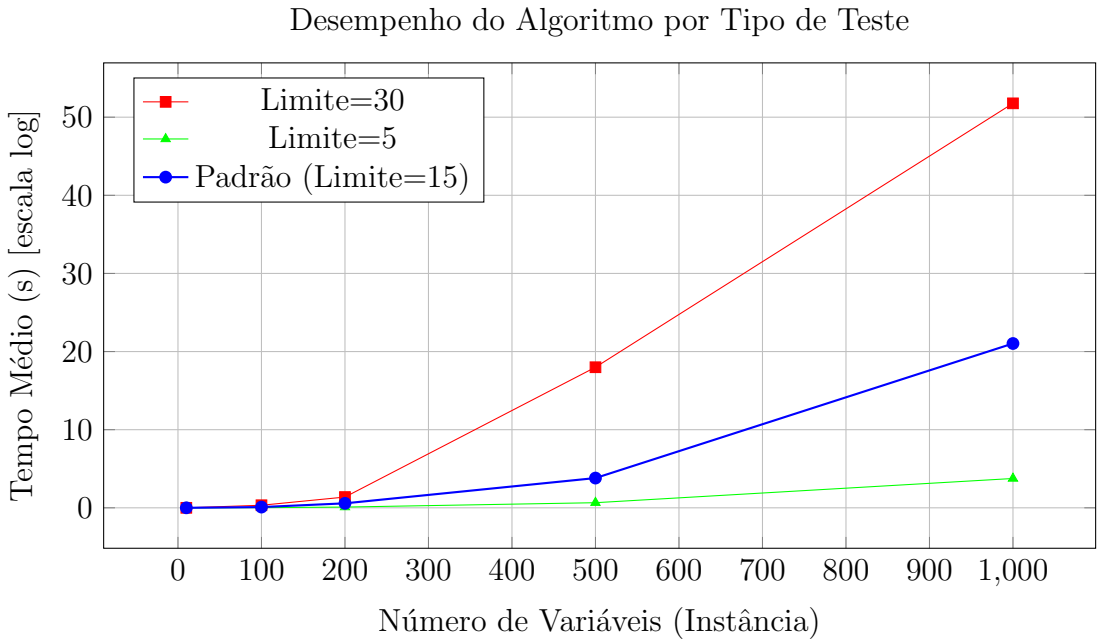


Figura 1: Comparação do tempo de execução.

não modela o **princípio da localidade**, um comportamento prevalente em software real.

No simulador, os usos de uma variável são distribuídos de forma semi-aleatória ao longo de todo o código. Este método pode gerar variáveis com *live ranges* longos e fragmentados, que se estendem por múltiplos escopos. Tal comportamento contrasta com o de programas típicos, onde o tempo de vida de uma variável é frequentemente restrito a uma única função ou a um laço de repetição.

Como consequência, os grafos de interferência gerados, embora válidos, podem apresentar uma topologia menos comum em cenários práticos. Portanto, os resultados obtidos são uma excelente avaliação da robustez da heurística em grafos complexos, mas a sua performance em códigos com alta localidade não foi avaliada devido a limitação da atual implementação do gerador.

# 9 Conclusão

A presente análise empírica do algoritmo de alocação de registradores de George-Appel, notável por sua complexa natureza iterativa, demonstrou que seu desempenho prático é criticamente ditado pelo *live range* das variáveis. Instâncias com *live ranges* longos resultaram em grafos de interferência densos, cujo custo computacional foi amplificado não apenas na coloração, mas crucialmente na fase de reescrita do programa (*spill*), que provoca uma explosão não linear no número de variáveis. Em contrapartida, *live ranges* curtos geraram grafos esparsos, processados com eficiência significativamente maior.

Reconhece-se que o simulador utilizado não modela o princípio da localidade, podendo gerar cenários de teste mais desafiadores que programas reais. Esta limitação aponta para trabalhos futuros, focados no aprimoramento do gerador de instâncias e na validação da heurística em *benchmarks* de compiladores estabelecidos. Em suma, o estudo valida que a eficiência do algoritmo de George-Appel está intrinsecamente ligada à topologia do grafo de interferência, um fator mais determinante que o número bruto de variáveis.

# Referências

astral-sh (s.d.). “uv”. URL: https://github.com/astral-sh/uv.

Briggs, Preston (abr. de 1992). “Register Allocation via Graph Colloring”. Em: URL: https://www.cs.utexas.edu/~mckinley/380C/lecs/briggs-thesis-1992.pdf.

Chaitin, G. J. (1982). “Register Allocation and Spilling Via Graph Coloring”. Em: New York, NY, USA. URL: https://web.eecs.umich.edu/~mahlke/courses/583f12/reading/chaitin82.pdf.

George-Appel LAL George, Andrew W. Appel (jun. de 1996). “Iterated Register Coalescing”. Em: Princeton University. URL: https://dl.acm.org/doi/pdf/10.1145/229542.229546.