



Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística


Rodolfo Pereira Araújo

**Estratégias de exploração de vizinhança com GPU para
problemas de otimização**

Rio de Janeiro
2018

Rodolfo Pereira Araújo

**Estratégias de exploração de vizinhança com GPU para problemas de
otimização**



Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Dr. Igor Machado Coelho

Coorientador: Prof. Dr. Leandro Augusto Justen Marzulo

Rio de Janeiro
2018

Rodolfo Pereira Araújo

**Estratégias de exploração de vizinhança com GPU para problemas de
otimização**

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em ?? de Setembro de 2018

Banca Examinadora:

Prof. Dr. Igor Machado Coelho (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Leandro Augusto Justen Marzulo (Coorientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dra. Cristiana Barbosa Bentes
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Tiago Assumpção de Oliveira Alves
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Luiz Satoru Ochi
Instituto de Computação - UFF

Rio de Janeiro
2018

AGRADECIMENTOS

Quero agradecer à Deus que me iluminou com as pessoas que colocou no meu caminho durante esta jornada e por toda minha vida.

Agradeço à minha esposa Izabel por todo apoio, carinho, companheirismo e amizade.

À minha mãe, pai e irmão e minha família que sempre foram os pilares na minha vida e realizações.

Aos meus orientadores, Igor e Marzulo, pelo empenho dedicado à elaboração deste trabalho e por confiarem em mim nesta empreitada. E a todos os professores por todo o conhecimento proporcionado para minha formação e que contribuiu para este trabalho.

RESUMO

ARAÚJO, Rodolfo Pereira. Estratégias de exploração de vizinhança com GPU para problemas de otimização. 2018. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Problemas de otimização são de grande importância para diversos setores da indústria, desde o planejamento de produção até escoamento e transporte dos produtos. Diversos problemas de interesse se enquadram na classe NP-Difícil, sendo desconhecidos algoritmos eficientes para resolvê-los de forma exata em tempo polinomial. Assim, estratégias heurísticas com capacidade de escapar de ótimos locais de baixa qualidade (meta-heurísticas) são geralmente empregadas. A busca local é, em geral, a etapa mais custosa, em termos de tempo computacional, do processo de uma meta-heurística, desta forma torna-se muito importante fazer bom uso dos recursos nela utilizados. Esta dissertação estuda o emprego de múltiplas estratégias de vizinhança utilizadas paralelamente para explorar um espaço de vizinhança maior e melhor aproveitar os recursos computacionais. O processamento paralelo das estratégias de vizinhança é implementado em nível de grão fino, através de processamento em GPU, e grão grosso, por meio de processamento multi core e processamento em rede, sendo os dois níveis combinados num ambiente heterogêneo, para arquiteturas von Neumann e dataflow.

Palavras-chave: Meta-heurística, Busca Local, Dataflow, Graphics Processing Unit, Variable Neighborhood Descent.

ABSTRACT

ARAÚJO, Rodolfo Pereira. Estratégias de exploração de vizinhança com GPU para problemas de otimização. 2018. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Optimization problems have big importance in the industry field, from production management to production outflow and product transportation. Many problems of interest are classified as NP-Hard, so there is no known algorithm to find the exact solution in a polynomial time. Therefore heuristic strategies with the ability to escape from poor quality local optima (meta-heuristics) are generally employed. In general, the local search is the most costly, in computational time, phase of a meta-heuristic, becoming mandatory a good use of the available resources. The parallel processing of neighborhood strategies is implemented at the fine grain level through GPU processing and coarse grain through multi-core processing and network processing, the combination of the two level parallelization in a heterogeneous environment for von Neumann architectures and dataflow.

Keywords: Meta-heuristics, Local Search, Dataflow, Graphics Processing Unit, Variable Neighborhood Descent.

LISTA DE ILUSTRAÇÕES

Figura 1 – Espaço de busca explorado por um método de otimização. São indicados mínimo local, mínimo global e vizinhança, conceitos que serão definidos a frente.	12
Figura 2 – Exemplo de conversão código para grafo de dependências, conforme apresentado em (MARZULO, 2011). O quadro A mostra um trecho de código e o quadro B o grafo dataflow associado.	14
Figura 3 – Exemplo de solução com as latências de cada cidade e cálculo da função objetivo	17
Figura 4 – Solução s_1	20
Figura 5 – Solução $m_1 \circ s_1$, com m_1 sendo 2-opt(1,7).	20
Figura 6 – Solução $m_2 \circ s_1$, com m_2 sendo 2-opt(5,6).	21
Figura 7 – Solução $m_3 \circ s_1$, com m_3 sendo 2-opt(1,3).	21
Figura 8 – Solução $m_1 \circ m_2 \circ s_1$, com m_1 sendo 2-opt(1,7) e m_2 sendo 2-opt(5,6).	23
Figura 9 – Solução $m_2 \circ m_1 \circ s_1$, com m_2 sendo 2-opt(5,6) e m_1 sendo 2-opt(1,7).	23
Figura 10 – Solução $m_2 \circ m_3 \circ s_1 = m_3 \circ m_2 \circ s_1$, com m_2 sendo 2-opt(5,6) e m_3 sendo 2-opt(1,3).	25
Figura 11 – Solução $m_1 \circ m_3 \circ s_1$, com m_1 sendo 2-opt(1,7) e m_3 sendo 2-opt(1,3).	25
Figura 12 – Solução $m_3 \circ m_1 \circ s_1$, com m_3 sendo 2-opt(1,3) e m_1 sendo 2-opt(1,7).	26
Figura 13 – Exemplo de conversão código para grafo de dependências. À esquerda pode ser visto um trecho de código em <u>python</u> ao passo que à direita é exibido o grafo dataflow associado.	30
Figura 14 – A Arquitetura da Sucuri (de (ALVES et al., 2014)). A mesma estrutura é replicada em cada nó, entretanto apenas no Scheduler do nó 0 do cluster contem o Matching Unit e o Ready Queue . Este é responsável por receber operandos dos workers locais e de outros Schedulers , além disso este gera as tarefas para serem enfileiradas no Ready Queue	31
Figura 15 – Pipelining com Sucuri. Painel A mostra um grafo de uma aplicação dataflow, o painel B descreve o grafo usando a Sucuri.	33
Figura 16 – Arquitetura simplificada do dataflow para o RVND com as vizinhanças utilizadas.	36
Figura 17 – Uma vizinhança e suas ligações ao grafo dataflow no RVND.	37
Figura 18 – Arquitetura simplificada do dataflow para o DVND com as vizinhanças utilizadas.	38
Figura 19 – Uma vizinhança e suas ligações ao grafo dataflow no DVND.	39
Figura 20 – FF identifica o nó de flip flop.	41
Figura 21 – Peça de um grafo dataflow em que o nó <i>MO</i> do dataflow possui múltiplas saídas.	41
Figura 22 – Quando o nó <i>MO</i> termina de processar seu resultado é enviado para todos os nós subsequentes (<i>next 0</i> , <i>next 1</i> e <i>next 2</i>).	42

Figura 23 – Quando o nó <i>MO</i> termina de processar é possível escolher qual porta de saída será utilizada e assim decidir o destino da informação.	42
Figura 24 – Exemplo de um grafo de Histórico. A combinação ótima de custos consiste nos movimentos m_2 , m_3 , m_5 e m_6	45
Figura 25 – Vizinhança com suas trocas para $n = 5$	52
Figura 26 – Vizinhança com suas trocas dividida para $n = 5$	52
Figura 27 – Tempo do DVND, <i>SOG</i> refere-se a uma porta de saída e <i>MOG</i> a múltiplas portas de saída, n indica o tamanho, m indica o número de máquinas utilizadas. Instâncias 0 a 3.	55
Figura 28 – Tempo do DVND, <i>SOG</i> refere-se a uma porta de saída e <i>MOG</i> a múltiplas portas de saída, n indica o tamanho, m indica o número de máquinas utilizadas. Instâncias 4 a 7.	56
Figura 29 – Tempos do RVND, n representa o tamanho da instância, m indica o número de máquinas, <i>RC</i> refere-se ao RVND clássico e <i>RD</i> ao RVND implementado em dataflow. Instâncias 0 a 3.	59
Figura 30 – Tempos do RVND, n representa o tamanho da instância, m indica o número de máquinas, <i>RC</i> refere-se ao RVND clássico e <i>RD</i> ao RVND implementado em dataflow. Instâncias 4 a 7.	60
Figura 31 – Melhoria no valor da solução para o RVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao RVND clássico e <i>DD</i> ao RVND implementado em dataflow. Instâncias 0 a 3.	61
Figura 32 – Melhoria no valor da solução para o RVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao RVND clássico e <i>DD</i> ao RVND implementado em dataflow. Instâncias 4 a 7.	62
Figura 33 – Tempo do DVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao DVND clássico e <i>DD</i> ao DVND implementado em dataflow. Instâncias 0 a 3.	66
Figura 34 – Tempo do DVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao DVND clássico e <i>DD</i> ao DVND implementado em dataflow. Instâncias 4 a 7.	67
Figura 35 – Melhoria no valor da solução do DVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao DVND clássico e <i>DD</i> ao DVND implementado em dataflow. Instâncias 0 a 3.	69
Figura 36 – Melhoria no valor da solução do DVND, n representa o tamanho da instância, m indica o número de máquinas, <i>DC</i> refere-se ao DVND clássico e <i>DD</i> ao DVND implementado em dataflow. Instâncias 4 a 7.	70
Figura 37 – Tempo dos algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 0 a 3.	74
Figura 38 – Tempo dos algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 4 a 7.	75
Figura 39 – Melhoria no valor da solução para os algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 0 a 3.	76
Figura 40 – Melhoria no valor da solução para os algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 4 a 7.	77

- Figura 41 – Tempo do DVND vs GDND, *DVND* refere-se ao tempo gasto pelo algoritmo de mesmo nome, para *GDVND* é análogo ao anterior, no caso do *GDVND-MAN* este se refere ao tempo do *GDVND* subtraído do tempo para gerenciar os movimentos, n representa o tamanho da instância, m indica o número de máquinas. Instâncias 0 a 3. 78
- Figura 42 – Tempo do DVND vs GDND, *DVND* refere-se ao tempo gasto pelo algoritmo de mesmo nome, para *GDVND* é análogo ao anterior, no caso do *GDVND-MAN* este se refere ao tempo do *GDVND* subtraído do tempo para gerenciar os movimentos, n representa o tamanho da instância, m indica o número de máquinas. Instâncias 4 a 7. 79

LISTA DE TABELAS

Tabela 1 – Informações de distâncias e localização das cidades para os exemplos apresentados.	19
Tabela 2 – Execução do GDVND	46
Tabela 3 – Consolidação de uma nova solução no GDVND	46
Tabela 4 – Configuração de lançamento para os kernels das vizinhanças. Onde <i>Grid</i> refere-se a quantidade de grides usada, <i>Block</i> o tamanho de cada bloco de threads e <i>Shared</i> indica a quantidade de memória compartilhada utilizada pelas threads em cada bloco.	50
Tabela 5 – Tabela de movimentos independentes (não conflitantes)	51
Tabela 6 – Tempos comparativos do SOG vs MOG onde SOG indica a execução com uma porta de saída e MOG com múltiplas portas de saída. Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	54
Tabela 7 – Tempos comparativos do RVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	57
Tabela 8 – Comparativos de melhoria na solução para o RVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	58
Tabela 9 – Tempos comparativos do DVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	65

Tabela 10 – Comparativos de melhoria na solução para o DVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	68
Tabela 11 – Tempos comparativos do GDVND com DVND e RVND. Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	71
Tabela 12 – Comparativos de melhoria na solução para o GDVND com DVND e RVND. Instância ($\#$), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).	72

LISTA DE ALGORITMOS

Algoritmo 1 – First Improvement para um problema de minimização	26
Algoritmo 2 – Best Improvement para um problema de minimização	26
Algoritmo 3 – Random Selection para um problema de minimização	27
Algoritmo 4 – Busca local definida de forma genérica	28
Algoritmo 5 – RVND clássico	33
Algoritmo 6 – DVND clássico	34
Algoritmo 7 – Nó de vizinhança do RVND	37
Algoritmo 8 – Nó <i>man</i> do DVND	38
Algoritmo 9 – Nó <i>man</i> do GDVND	43
Algoritmo 10 – Combinando movimentos de soluções diferentes	44

SUMÁRIO

1	Introdução	12
1.1	Motivação	15
1.2	Objetivos	15
1.3	Estrutura deste documento	15
2	Revisão de literatura e conceituação teórica	16
2.1	Problemas de Decisão	16
2.2	Problemas de Otimização	16
2.3	Problema da Mínima Latência	17
2.4	Busca local	18
2.4.1	Vizinhança	19
2.4.2	Movimento	19
2.4.2.1	Movimentos livres de estrutura – <u>structure-free moves</u>	20
2.4.2.2	Movimentos fracamente independentes – <u>weakly independent moves</u>	22
2.4.2.3	Movimentos independentes	22
2.4.2.4	Movimentos fortemente independentes – <u>strongly independent moves</u>	24
2.4.2.5	Movimentos conflitantes	25
2.4.3	First Improvement vs Best Improvement	26
2.4.4	Random Selection	27
2.4.5	Multi Improvement	27
2.4.6	Passo iterativo	27
2.5	Ótimo global vs Ótimo local	28
2.6	Meta-heurísticas	28
2.7	Dataflow	29
2.7.1	Sucuri	29
2.8	RVND	33
2.9	DVND	34
3	Metodologia proposta	36
3.1	RVND	36
3.1.1	Passo iterativo	37
3.2	DVND	37
3.2.1	Passo iterativo	40
3.2.2	Nó de flip flop	40
3.2.3	Múltiplas portas de saída	40
3.3	GDVND	43
3.3.1	Deteção movimentos independentes	44
3.3.2	Exemplo de execução	45
3.3.3	Controle de execução entre CPU e GPU	46
3.3.4	Passo iterativo	47
3.4	Vizinhanças	48
3.4.1	Tabela de conflitos	49

3.5	Decomposição de vizinhanças (Disaggregated Neighborhoods)	52
4	Resultados	53
4.1	Instâncias	53
4.2	Implementação e ambiente computacional	53
4.3	Múltiplas portas de saída	54
4.4	RVND	57
4.4.1	Tempo	57
4.4.2	Melhoria no valor da solução	57
4.5	DVND	63
4.5.1	Tempo	63
4.5.2	Melhoria no valor da solução	63
4.6	GDVND	71
4.6.1	Tempo	71
4.6.2	Melhoria no valor da solução	71
4.6.3	Analisando o tempo para combinar movimentos	72
5	Conclusões	80
5.1	RVND	80
5.2	DVND	80
5.3	GDVND	80
5.4	Propostas futuras	81
5.4.1	Testar com instâncias maiores	81
5.4.2	Decomposição de vizinhanças	81
	Referências	82

1 INTRODUÇÃO

Muitos dos problemas encontrados no dia-a-dia podem possuir inúmeras soluções, que em geral detêm um fator de satisfação associado (e.g.: lucro obtido, custo de utilização). Nessa linha, o ramo da otimização atua no estudo destes de modo a buscar minimizar ou maximizar o valor da função objetivo (satisfação) associada.

Conforme pode ser visto na Figura 1, um método de otimização explora o espaço de busca a procura da solução que apresente o melhor valor de função objetivo.

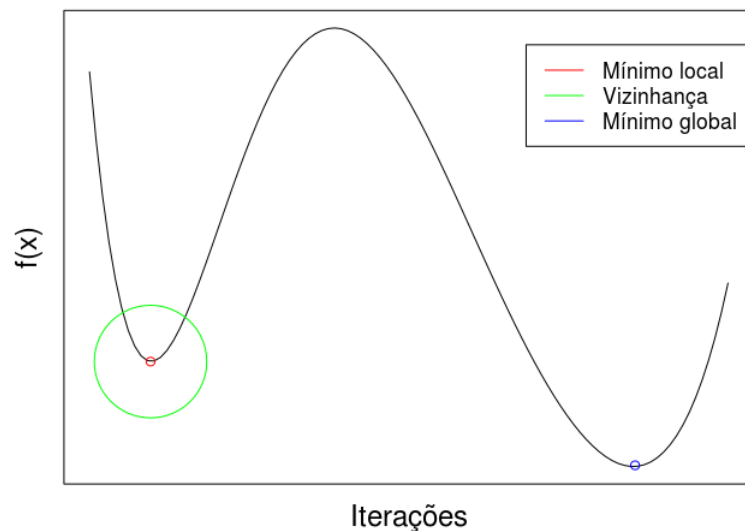


Figura 1 – Espaço de busca explorado por um método de otimização. São indicados mínimo local, mínimo global e vizinhança, conceitos que serão definidos a frente.

Quando é necessário resolver problemas de otimização, um campo promissor é o das meta-heurísticas, contudo estes algoritmos podem demandar muito tempo de processamento, especialmente ao se resolver problemas com grandes instâncias. Isto posto, podemos ressaltar a importância de utilizar métodos eficientes para resolução de tais problemas.

Uma parte importante no universo das meta-heurísticas são os algoritmos de busca que, inicialmente, recebem uma solução para um problema e rastreiam seu espaço de busca para então retornar a melhor solução encontrada. O espaço de busca é um super conjunto de algumas estratégias de vizinhança para a solução atual e as que surgirem.

Uma boa alternativa para melhorar o tempo de processamento destes problemas é pela utilização de programação paralela (GENDREAU; POTVIN, 2010). A maioria dos algoritmos de otimização até então projetados foram feitos para funcionarem de forma “naturalmente” sequencial, criando uma árdua tarefa, muitas vezes hercúlea, para os programadores que desejam alterar a implementação do algoritmo para rodar de forma paralela. A abordagem mais utilizada para paralelização de meta-heurísticas consiste em

escolher métodos ou partes do método que podem ser executadas independentemente e lançar sua execução paralela (Bag-of-Tasks). Não obstante apenas alguns trabalhos científicos são dedicados a realmente re-projetar estes algoritmos para aproveitar o poder de arquiteturas paralelas de forma profunda (RIOS et al., 2017b; RIOS et al., 2017a; ARAUJO et al., 2018).

Em abordagens tradicionais para paralelização de aplicações são empregadas arquiteturas Multiple Instruction Multiple Data (MIMD) (FLYNN, 1972), onde cada elemento de processamento possui streams de instruções e dados independentes. Desta forma, o desenvolvedor pode particionar as tarefas de sua aplicação para que sejam executadas em threads ou processos, mapeadas nos elementos de processamento. A comunicação entre tarefas pode ser realizada por memória compartilhada (multiprocessadores ou multicores) ou por troca de mensagens (como em clusters de computadores). Estes processadores seguem o modelo de Von Neumann, no qual a execução de uma instrução é guiada pelo fluxo de controle, de forma que a ordem das instruções no programa prescreve o que o processador deve fazer passo a passo. Este modelo assume que um *program counter* é usado para indicar a próxima instrução a ser executada e estas podem alterar o estado da máquina ao alterar valores de uma estrutura de armazenamento global, como um banco de registradores, cache ou memória principal.

Trabalhos recentes em modelos e linguagens de programação paralela (ALVES et al., 2011; DURAN et al., 2011; BALAKRISHNAN; SOHI, 2006; GUPTA; SOHI, 2011; TBB..., accessed on August 8, 2014.; BOSILCA et al., 2012; GIORGI et al., 2014) geraram novos frameworks para o modelo de programação dataflow, em diferentes níveis de abstração e granularidade, como solução para representar programas considerando as operações como ponto de vista principal, permitindo assim uma modelagem mais fácil de muitos problemas ao alta performance. Está claro que em alguns casos o esforço de programação (medido em termos de tempo de programação ou linhas de código) é consideravelmente reduzido quando se utiliza programação dataflow para paralelizar as aplicações, se compararmos com as ferramentas tradicionais como OpenMP ou Pthreads. Além disso, o desempenho da execução do aplicações desenvolvidas conforme o modelo dataflow pode ser comparado com métodos tradicionais (ALVES et al., 2011; ALVES; MARZULO; FRANCA, 2013).

A Figura 2 mostra um paralelo entre a versão Von Neumann, no quadro A e a implementação do mesmo código no modelo dataflow, exemplificada no quadro B. Pelo quadro B pode-se perceber que a execução das operações $x + y$ e $k * j$ podem ser executadas em qualquer ordem, ou mesmo simultaneamente, sem alterar o resultado final da execução.

Usualmente a programação dataflow é feita instanciando trechos de código e os conectando em um grafo de acordo com suas dependências de dados, livrando assim o programador de grande parte da complexidade da programação paralela uma vez que esta, bem como a sincronização, são realizados pelo ambiente de execução dataflow. O modelo dataflow expõe naturalmente o paralelismo, instruções são executadas conforme suas dependências de dados (na ordem do fluxo de dados, dataflow), i.e., instruções são disparadas assim que seus operandos estão disponíveis, dessa forma o desafio passa a ser modelar o grafo de dependências das operações tendo como uma importante decisão a se tomar o tamanho do grão de cada nó do grafo de forma a comportar o paralelismo sem sobrecarregar o algoritmo podendo assim causar um overhead desnecessário na troca de mensagens.

Problemas NP-Difíceis (GAREY; JOHNSON, 1990) surgem de cenários práticos, como o roteamento de um conjunto de veículos para entregas e coletas, ou visitar um conjunto de

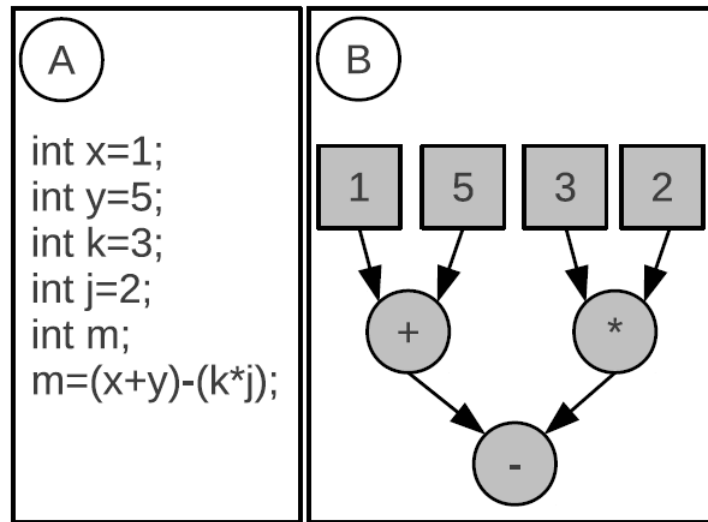


Figura 2 – Exemplo de conversão código para grafo de dependências, conforme apresentado em (MARZULO, 2011). O quadro A mostra um trecho de código e o quadro B o grafo dataflow associado.

idades percorrendo a menor rota possível. Este último sendo conhecido como Problema do Caixeiro Viajante (PCV), um dos mais importantes (e não resolvidos) problemas no campo da ciência da computação. Devido a sua combinatoriedade natural, não existe algoritmo conhecido que resolva o PCV em tempo polinomial. Todavia, muitos algoritmos heurísticos e combinações de métodos exatos com métodos heurísticos são capazes de resolver o PCV para instâncias com milhares de cidades. Como uma variante menos explorada do PCV, consideramos o Problema da Mínima Latência (PML), que é uma variante do PCV onde todos os nós precisam ser visitados, mas o custo é o somatório do acumulado das distâncias ponto a ponto.

Muitos trabalhos recentes produziram algoritmos que obtiveram resultados eficientes utilizando RVND (SOUZA et al., 2010; SILVA et al., 2012; SUBRAMANIAN; UCHOA; OCHI, 2013), cuja ideia principal é que ao se atingir um mínimo local para uma estratégia de vizinhança ainda pode existir um vizinho em outra vizinhança com valor de função objetivo melhor que o já encontrado, o que motivou a implementação de uma versão distribuída. O DVND, proposto em (RIOS et al., 2018), é uma versão paralela do método de busca VND, adotando-se uma visão dataflow para o algoritmo é possível uma implementação conforme a encontrada em (ARAUJO; COELHO; MARZULO, 2018). A arquitetura multi-core do DVND torna possível explorar diferentes vizinhanças simultaneamente (com diferentes implementações de busca local), e escolher o melhor resultado dos métodos de busca. Não obstante, os autores que obtiveram o estado da arte nos resultados para o PML não chegaram a estender a implementação do DVND para múltiplos e independentes nós de processamento devido a problemas de performance, serialização de dados e controle de fluxo utilizando a tecnologia MPI (RIOS et al., 2017a). Por esse motivo esse trabalho propõe uma nova implementação do DVND capaz de lidar com múltiplos nodos de maneira mais natural, modelando o método como um grafo dataflow e usando o framework em Python para dataflow Sucuri (ALVES et al., 2014).

1.1 Motivação

Inúmeras aplicações práticas podem utilizar meta-heurísticas para resolver problemas de otimização, sendo comum para muitos desses processos sua etapa final ser constituída de um algoritmo de busca local. Este por sua vez faz uso de estratégias de vizinhança para enumerar o espaço de busca. Versões clássicas do processos de busca local utilizam uma vizinhança em conjunto com uma estratégia de exploração (Primeira melhora, Melhor vizinho).

Poucos são os trabalhos da literatura que combinam estratégias de vizinhança diferentes como em Variable Neighborhood Descent (VND) e mais raros são os trabalhos que usam diferentes estratégias de exploração de vizinhança como em (RIOS et al., 2017b).

1.2 Objetivos

Este estudo visa propor estratégias diferenciadas de solução de problemas de otimização, para tanto podemos enumerar os seguintes objetivos:

- A investigação do uso de estratégias distintas de vizinhanças almeja a realização de uma busca mais extensa no espaço de soluções, contudo isto aumenta sua complexidade computacional, o que abre caminho para o item seguinte;
- Propor algoritmos paralelos que se beneficiam de tecnologias emergentes, porém ainda pouco exploradas no ramo da pesquisa operacional, que é a utilização de aceleração pelo uso de placas gráficas;
- Utilizar melhor os recursos usados no processamento realizado ao aproveitar computações realizadas em nós diferentes.

O objetivo final é propor um método que seja capaz de fazer uso de múltiplas estratégias de vizinhança aproveitando recursos de paralelização de grão fino e grão grosso sem precisar acoplar a solução a um problema específico.

1.3 Estrutura deste documento

O restante deste trabalho é organizado conforme o segue:

- Capítulo 2: descreve os fundamentos teóricos e termos utilizados nessa dissertação, no desenvolvimento deste capítulo são caracterizados e ilustrados os termos e meta-heurísticas utilizados por descrições independentes;
- Capítulo 3: descreve e apresenta os algoritmos propostos para resolver o Problema da Mínima Latência mas não se limitando a este problema. Este capítulo detalha os métodos bem como seus componentes e pseudocódigos;
- Capítulo 4: mostra os resultados computacionais deste trabalho. Os métodos Data-flow DVND e GDVND propostos podem utilizar diferentes estratégias e da mesma forma podem ser aplicados a problemas variados;
- Capítulo 5: apresenta as conclusões do trabalho desenvolvido nessa dissertação com as propostas de trabalhos futuros.

2 REVISÃO DE LITERATURA E CONCEITUAÇÃO TEÓRICA

Este capítulo tem como objetivo introduzir os conceitos teóricos básicos utilizados no documento e bases para apresentar a método.

2.1 Problemas de Decisão

Um problema Π é dito um problema de decisão quando seu conjunto solução é composto apenas pelos elementos *Sim* e *Não*, ou seja:

$$\begin{aligned}\Pi : D &\rightarrow Im \\ Im &= \{Sim, Não\}\end{aligned}\tag{2.1}$$

São exemplos de problemas de decisão:

- Seja $x \in C$, sendo x um número pertencente ao conjunto C , x é o menor número deste conjunto?
- Seja um grafo $G(V,A)$ denotado pelos arestas A e vértices V , existe um caminho do vértice x para o vértice y com custo menor que c ?

2.2 Problemas de Otimização

Seja Π um problema, S o conjunto de soluções viáveis para o mesmo e f a função objetivo que associa uma solução s_i a um valor numérico então temos:

$$\begin{aligned}S &= \{s_1, s_2, \dots, s_n\} \\ f : S &\rightarrow \mathbb{R}\end{aligned}\tag{2.2}$$

Um problema de otimização, em geral, pode ser de minimização ou de maximização. Π é um problema de minimização se ele consiste em determinar uma solução s^* tal que:

$$s^* \in S \mid f(s^*) \leq f(s), \forall s \in S\tag{2.3}$$

De forma análoga um problema de maximização pode ser dado por:

$$s^* \in S \mid f(s^*) \geq f(s), \forall s \in S\tag{2.4}$$

Os problemas de otimização podem ser divididos em dois tipos:

- Otimização contínua: nesse tipo de problema pelo menos uma das variáveis x do conjunto de variáveis X pode assumir infinitos valores;

- Otimização combinatória: problemas em que toda variável x no conjunto de variáveis X é discreta, podendo assumir apenas um número finito ou infinito porém contável de valores.

Desta forma, existe um número finito de soluções viáveis para um problema de otimização combinatória. Para todo problema de decisão existe um problema de otimização associado, tomemos com exemplo os problemas da seção 2.1 e teremos os seguintes problemas de otimização:

- Seja um conjunto C , encontrar $x \in C \mid x < y, \forall y \in C$.
- Seja um grafo $G(V,A)$ denotado pelas arestas A e vértices V , encontrar o menor caminho do vértice x para o vértice y .

2.3 Problema da Mínima Latência

O Problema da Mínima Latência (PML) é um problema de otimização, sendo uma variante do PCV no qual o objetivo é minimizar o tempo de chegada (ou latência) aos vértices, e não a distância ou tempo da rota como no problema original. O PML pode ser definido como um grafo direcionado $G = (V,E)$, onde $V = \{0,1,\dots,n\}$ é um conjunto de vértices e $E = \{(i,j) : i,j \in V, i \neq j\}$ um conjunto de arestas que conectam os vértices. Para cada arco (i,j) existe um tempo de viagem associado igual a $t(i,j)$. O vértice 0 representa o ponto de saída (depósito) e os demais os clientes a serem visitados. O tempo de chegada (ou latência) a um cliente $i \in V$, é denotado por $l(i)$, o qual é definido pelo tempo de viagem do depósito até o vértice i .

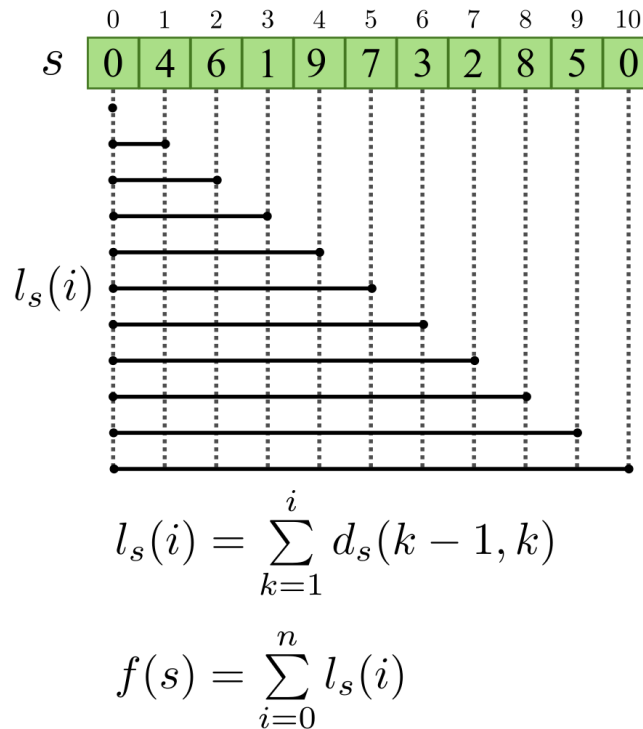


Figura 3 – Exemplo de solução com as latências de cada cidade e cálculo da função objetivo

O objetivo do PML é, iniciando do depósito, determinar o ciclo Hamiltoniano s que minimize a latência expressa por $L(s) = \sum_{i=0}^n l(i)$ como pode ser visto na Figura 3. Assim sendo, uma solução viável do PML consiste numa permutação de n clientes determinando a ordem de visita dos mesmos. Tomemos o exemplo a seguir, se $n = 9$, $s = [0, 8, 3, 7, 1, 4, 2, 5, 6, 0]$ é uma solução viável para o PML (de fato, qualquer permutação 1..8 começando e terminando no vértice zero é uma solução viável).

Apesar da formulação simples e de sua grande aplicação na otimização de latência em redes, o PML é um problema complexo, sendo provado que o PML é NP-Difícil (SILVA et al., 2012). Apesar da semelhança na formulação do PML com a do Problema do Caixeiro Viajante (PCV) a sua função objetivo é mais complexa de ser calculada que a do PCV. No PML, pequenas alterações no vetor solução podem levar a grandes alterações no resultado final da solução e a natureza não local da função objetivo faz com que uma simples inserção afete todas as latências subsequentes. Na literatura, o PML também é conhecido Problema do Caixeiro Viajante Cumulativo (BIANCO; MINGOZZI; RICCIARDELLI, 1993), Problema do Entregador (MLADENović; UROŠEVIĆ; HANAFI, 2013) e Problema do Reparador Viajante. (TSITSIKLIS, 1992).

Em trabalhos recentes, um procedimento de busca local baseado em Graphics Processing Unit (GPU) e computação multi-core foi proposto para o PML (RIOS et al., 2016). A ideia foi chamada Distributed Variable Neighborhood Descent (DVND), tentando explorar diferentes estratégias de vizinhança simultaneamente para uma solução de entrada.

Em otimização, uma vizinhança é definida como um conjunto de operações chamados "movimentos", que são capazes de realizar pequenas alterações na solução de entrada. Estas alterações podem ser, por exemplo, trocar dois elemento na permutação inicial, gerando dessa forma $\mathcal{O}(N^2)$ diferentes soluções (para o caso de uma permutação de tamanho N). Existe na literatura muitas dessas vizinhanças (como 2-Opt, OrOpt-1, OrOpt-2, Swap, ..., etc), conquanto por limitações computacionais estes são sempre explorados de forma sequencial, chamados de Variable Neighborhood Descent (VND). Com o objetivo de encontrar um ótimo local para o PML, a ideia principal do DVND é usar GPU para obter operações de grão fino (que em geral são rápidas) e explorar toda a vizinhança $\mathcal{O}(N^2)$ mais rápido que em CPU (como explicado em (RIOS et al., 2016)) e combinar as respostas das buscas, escolhendo a nova melhor solução.

Pelo exemplo na Figura 3 podemos ver que o valor da latência $L(s)$ será dado pelos somatórios das latências de todas as cidades, sendo $d_s^{i,j}$ a distância da cidade i para a cidade j na solução s , então temos:

$$L(s) = \sum_{i=0}^n l_s(i)$$

$$L(s) = l_s(0) + l_s(1) + l_s(2) + l_s(3) + l_s(4) + l_s(5) + l_s(6) + l_s(7) + l_s(8) + l_s(9)$$

$$L(s) = 10d_s^{0,4} + 9d_s^{4,6} + 8d_s^{6,1} + 7d_s^{1,9} + 6d_s^{9,7} + 5d_s^{7,3} + 4d_s^{3,2} + 3d_s^{2,8} + 2d_s^{8,5} + d_s^{5,0}$$

2.4 Busca local

Um algoritmo de busca local percorre iterativamente o espaço de soluções de um determinado problema melhorando a solução atual. Para tanto procura na vizinhança (definida na Seção 2.4.1) atual por uma solução melhor que a atual e repete o processo até não ser encontrada uma melhora.

2.4.1 Vizinhaça

Seja S o espaço de soluções de um problema de otimização Π , e $s \in S$ uma solução qualquer, considere a função objetivo $f : S \rightarrow \mathbb{R}$ que atribui um valor para cada solução. Denotamos então por $N^x(s)$ o conjunto de soluções vizinhas de s para a vizinhaça x com $N^x(s) \subseteq S$, em que as soluções dessa vizinhaça podem ser obtidas de s a partir da aplicação de determinadas operações. Uma solução s' é vizinha de s (isto é $s' \in N^x(s)$) segundo uma vizinhaça s se s' é alcançável a partir de s fazendo uso de uma pequena perturbação nesta última.

2.4.2 Movimento

Considere $m^k : S \rightarrow S$ a função que representa um movimento que leva uma solução s a uma solução s' ou de forma equivalente $s' = m^k \circ s$. Designamos então o movimento $m^k \in M^k$ no conjunto de movimentos da vizinhaça k .

Adicionalmente podemos definir o custo do movimento $m^k(s)$ em relação a solução s como sendo a diferença entre o valor da solução s' gerada ao aplicar este movimento à solução s e o valor da solução s , conforme pode ser visto na Equação 2.5, iremos omitir a função f e a vizinhaça k quando estiverem claros no contexto, desta forma a podemos simplificar a notação conforme a Equação 2.6. Formalmente, a notação circunflexo representa a função $\widehat{m} : S \rightarrow \mathbb{R}$.

$$\widehat{m}_f^k(s) = f(s') - f(s) \quad (2.5)$$

$$\widehat{m}_f^k(s) = \widehat{m}(s) = f(s') - f(s) \quad (2.6)$$

Para exemplificar as definições que se seguem tomemos a Tabela 1, consideremos um conjunto de cidades com suas localizações conforme a Tabela 1a. Para facilitar o acompanhamento considere a matriz de distâncias expressa na Tabela 1b.

Cidade	x	y		A	B	C	D	E	F	G	H	I
A	200	200	A		224	600	728	640	806	825	510	300
B	100	400	B	224		412	583	583	825	900	671	447
C	200	800	C	600	412		224	412	728	894	860	671
D	400	900	D	728	583	224		283	583	781	854	707
E	600	700	E	640	583	412	283		316	500	608	510
F	900	600	F	806	825	728	583	316		224	539	566
G	1000	400	G	825	900	894	781	500	224		424	539
H	700	100	H	510	671	860	854	608	539	424		224
I	500	200	I	300	447	671	707	510	566	539	224	

(a) Coordenadas das cidades.

(b) Matriz de distâncias euclidianas com valores inteiros arredondados.

Tabela 1 – Informações de distâncias e localização das cidades para os exemplos apresentados.

Tomemos como exemplo a solução s_1 expressa na Figura 4, este apresenta uma solução para o problema apresentado na Tabela 1 com valor de função objetivo 2631 para o PCV e 13074 para o PML.

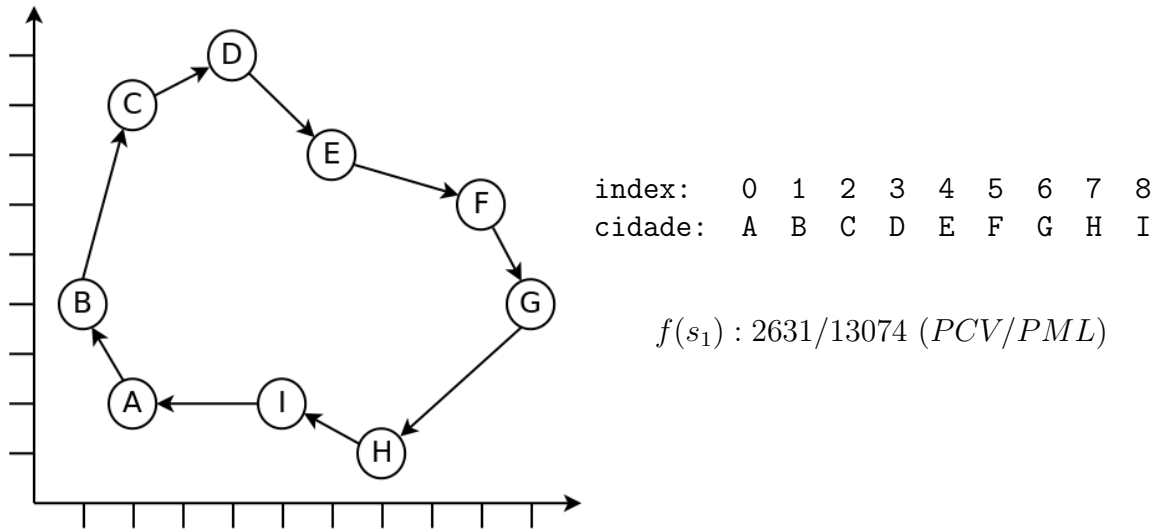


Figura 4 – Solução s_1 .

Ao aplicarmos à solução anterior s_1 o movimento m_1 2-opt(1,7), exibido na Figura 5, este reverte bloco de 1 a 7 (remove implicitamente duas arestas e reconecta a rota), o custo do movimento é $\widehat{m}_1(s_1)$: 509 para o PCV e 3113 para o PML.

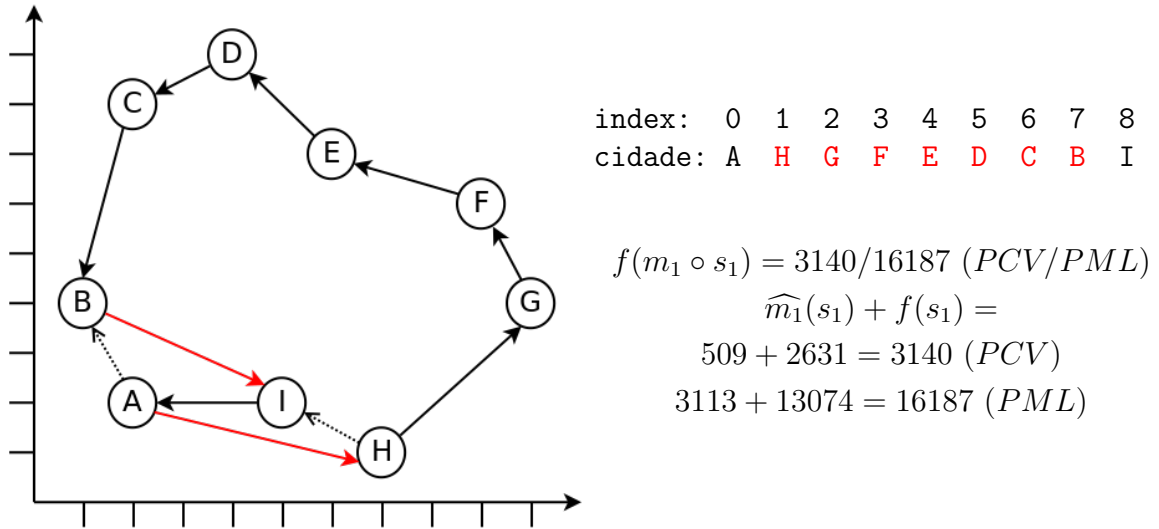


Figura 5 – Solução $m_1 \circ s_1$, com m_1 sendo 2-opt(1,7).

Outra opção seria escolher o movimento m_2 2-opt(5,6) para ser aplicado à solução, este reverte bloco de 5 a 6 (remove implicitamente duas arestas e reconecta rota), neste caso o custo do movimento é $\widehat{m}_2(s_1)$: 299/1265 para o PCV e PML respectivamente, em detalhes na Figura 6.

Adicionalmente se pode ver que a Figura 7 mostra o movimento m_3 2-opt(1,3) sendo aplicado à solução s_1 , este reverte bloco de 1 a 3 (remove implicitamente duas arestas e reconecta rota) e apresenta um custo de $\widehat{m}_3(s_1)$: 804/6148 para o PCV e PML.

2.4.2.1 Movimentos livres de estrutura – structure-free moves

Quando a estrutura de armazenamento interna utilizada para representar uma solução permite a aplicação de um movimento para qualquer solução temos um movimento livre

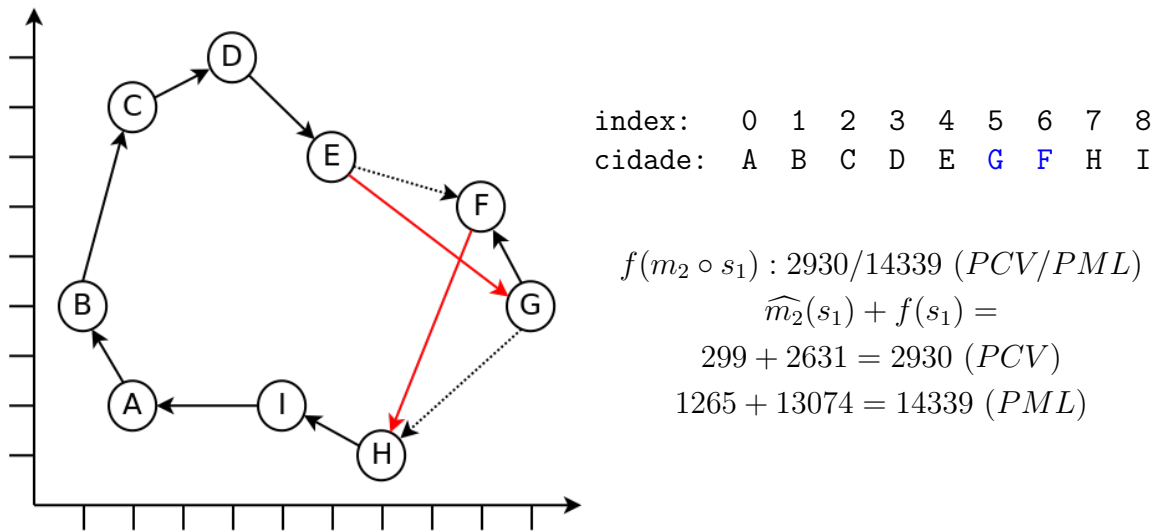


Figura 6 – Solução $m_2 \circ s_1$, com m_2 sendo 2-opt(5,6).

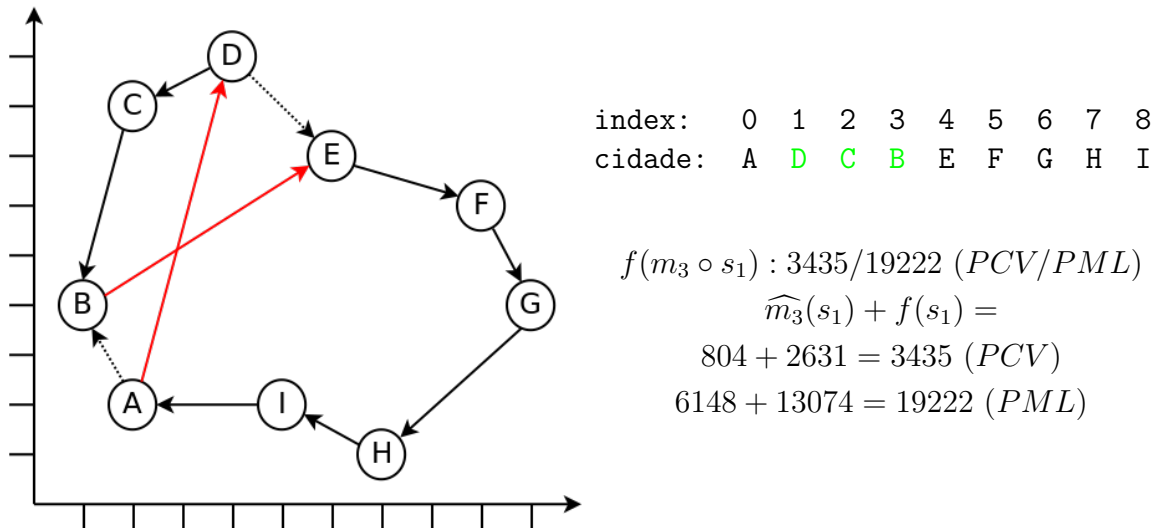


Figura 7 – Solução $m_3 \circ s_1$, com m_3 sendo 2-opt(1,3).

de estrutura. Em outras palavras, um movimento m é livre de estrutura se ele sempre pode ser aplicado a uma solução s sem gerar uma solução inválida.

$$m \text{ é livre de estrutura} \iff m \circ s \in S, \forall s \in S \quad (2.7)$$

Quando uma classe de movimentos é livre de estrutura para um determinado problema então se pode dizer que a função a representar este movimento é fechada para o conjunto de soluções S .

A caracterização de um movimento como livre de estrutura depende de suas restrições e da sua representação. Desta forma, para o Problema do Caixeiro Viajante em que o grafo com as distâncias entre as cidades é completo, o movimento *Swap* será livre de estrutura contudo num grafo incompleto uma aplicação do *Swap* pode gerar uma solução inviável pois pode não existir um determinado percurso após a alteração na solução.

2.4.2.2 Movimentos fracamente independentes – weakly independent moves

Dois movimentos $\{m_2, m_3\} \subseteq \mathcal{I}$ são fracamente independentes quando a aplicação de um deles não altera o custo do outro movimento para apenas um dos movimentos. Formalmente temos que dois movimentos m_1 e m_2 são parcialmente independentes, ou seja $m_1 \parallel_p m_2$, se e somente se:

$$m_1 \parallel_p m_2 \iff \widehat{m}_1(s) = \widehat{m}_1(m_2 \circ s) \quad (2.8)$$

Corolário 1. *Se dois movimentos são parcialmente independentes então o valor da solução resultante será a soma do valor da solução inicial com o custo dos movimentos.*

Formalmente temos:

$$m_1 \parallel_p m_2 \implies f(m_1 \circ m_2 \circ s) = \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad (2.9)$$

Demonstração. Pela definição de custo do movimento (Equação 2.5) temos:

$$\begin{aligned} f(m_1 \circ m_2 \circ s) &= \widehat{m}_1(m_2 \circ s) + f(m_2 \circ s) \text{ Aplicando Equação 2.5} \\ f(m_1 \circ m_2 \circ s) &= \widehat{m}_1(m_2 \circ s) + \widehat{m}_2(s) + f(s) \end{aligned}$$

Aplicando-se a definição de movimentos parcialmente independentes temos que:

$$\begin{aligned} \widehat{m}_1(m_2 \circ s) &= \widehat{m}_1(s) \text{ Logo} \\ f(m_1 \circ m_2 \circ s) &= \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) + f(s) \end{aligned}$$

□

Note que a definição não permite comutatividade, logo nada pode ser afirmado para o caso da solução $m_2 \circ m_1 \circ s$.

Acompanhando o exemplo, se movimentos m_1 e m_2 forem parcialmente independentes temos:

$$\begin{aligned} (m_1 \circ m_2 \circ s_1) &= 3439/17452 \\ (m_1 \circ m_2 \circ s_1) &= \widehat{m}_1(s_1) + \widehat{m}_2(s_1) + f(s_1) = \end{aligned}$$

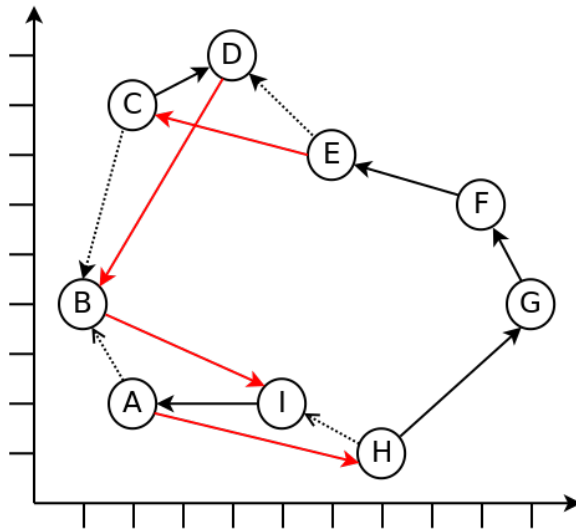
$$\begin{aligned} (m_1 \circ m_2 \circ s_1) &= \\ 509 + 299 + 2631 &= 3439 \text{ (PCV)} \\ 3113 + 1265 + 13074 &= 17452 \text{ (PML)} \end{aligned}$$

Conforme supracitado e ilustrado nas Figuras 8 e 9 podemos ver que a solução $m_2 \circ m_1 \circ s_1$ não verifica as mesmas propriedades pois apenas m_1 não é alterado por m_2 , desta forma estes movimentos são parcialmente independentes.

De fato podemos verificar pela Figura 9 que $\widehat{m}_2(s_1) = 299/1265$ é diferente de $\widehat{m}_2(m_1 \circ s_1) = 300/1158$.

2.4.2.3 Movimentos independentes

Movimentos independentes são aqueles que a aplicação de um não altera o valor do outro, o valor do movimento m_2 aplicado à solução $m_1 \circ s$ é igual ao valor deste aplicado à solução s , a independência de movimentos pode ser definida para movimentos de



index:	0	1	2	3	4	5	6	7	8
cidade:	A	H	F	G	E	D	C	B	I

$$f(m_1 \circ m_2 \circ s_1) = 3439/18211 \text{ (PCV/PML)}$$

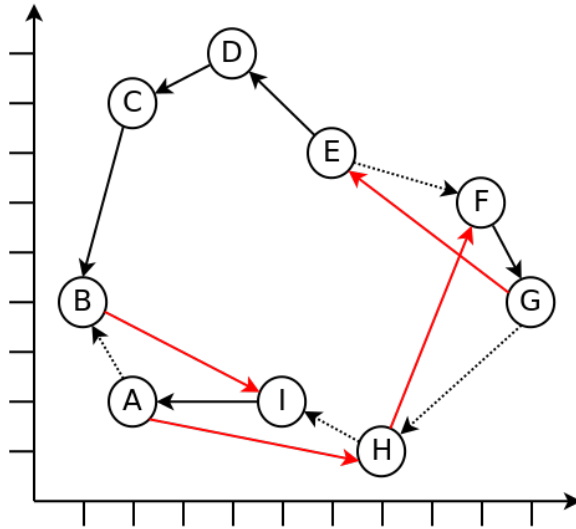
$$f(m_1 \circ m_2 \circ s_1) =$$

$$\widehat{m}_1(m_2 \circ s_1) + \widehat{m}_2(s_1) + f(s_1) =$$

$$509 + 299 + 2631 = 3439 \text{ (PCV)}$$

$$3872 + 1265 + 13074 = 18211 \text{ (PML)}$$

Figura 8 – Solução $m_1 \circ m_2 \circ s_1$, com m_1 sendo 2-opt(1,7) e m_2 sendo 2-opt(5,6).



index:	0	1	2	3	4	5	6	7	8
cidade:	A	H	G	F	E	C	D	B	I

$$f(m_2 \circ m_1 \circ s_1) = 3440/17345 \text{ (PCV/PML)}$$

$$f(m_2 \circ m_1 \circ s_1) =$$

$$\widehat{m}_2(m_1 \circ s_1) + \widehat{m}_1(s_1) + f(s_1) =$$

$$300 + 509 + 2631 = 3440 \text{ (PCV)}$$

$$1158 + 3113 + 13074 = 17345 \text{ (PML)}$$

Figura 9 – Solução $m_2 \circ m_1 \circ s_1$, com m_2 sendo 2-opt(5,6) e m_1 sendo 2-opt(1,7).

vizinhanças diferentes. Em outras palavras movimentos independentes são parcialmente independentes para ambos os lados. Formalmente temos que dois movimentos m_1 e m_2 são independentes $m_1 \parallel m_2$ se e somente se:

$$m_1 \parallel m_2 \iff \widehat{m}_1(m_2 \circ s) = \widehat{m}_1(s) \quad \wedge \quad \widehat{m}_2(m_1 \circ s) = \widehat{m}_2(s) \quad (2.10)$$

Teorema 2. *Dois movimentos m_1 e m_s são independentes então o valor da solução final não se altera ao alternar a ordem de aplicação dos movimentos, ou seja:*

$$m_1 \parallel m_2 \implies f(m_1 \circ m_2 \circ s) = f(m_2 \circ m_1 \circ s) \quad (2.11)$$

Demonstração. Suponhamos que m_1 e m_2 sejam independentes $\widehat{m}_1(m_2 \circ s) = \widehat{m}_1(s) \wedge \widehat{m}_2(m_1 \circ s) = \widehat{m}_2(s)$ mas $f(m_1 \circ m_2 \circ s) \neq f(m_1 \circ m_1 \circ s)$.

$$f(m_1 \circ m_2 \circ s) \neq f(m_2 \circ m_1 \circ s) \quad \text{De (2.5)}$$

$$\widehat{m}_1(m_2 \circ s) + f(m_2 \circ s) \neq \widehat{m}_2(m_1 \circ s) + f(m_1 \circ s) \quad \text{De (2.5)}$$

$$\widehat{m}_1(m_2 \circ s) + \widehat{m}_2 + f(s) \neq \widehat{m}_2(m_1 \circ s) + \widehat{m}_1 + f(s) \quad \text{Como } m_1 \parallel m_2$$

$$\widehat{m}_1(s) + \widehat{m}_2 + f(s) \neq \widehat{m}_2(s) + \widehat{m}_1 + f(s) \quad \text{Logo uma contradição } \perp$$

Assim, por *reductio ad absurdum*, temos que se m_1 e m_2 são independentes então $f(m_1 \circ m_2 \circ s) = f(m_2 \circ m_1 \circ s)$. \square

Outra propriedade interessante pode ser vista no teorema apresentado a seguir:

Teorema 3 (Teorema da independência dos movimentos dois a dois). *Se $m_1 \parallel m_2$ então o valor da solução gerada pela aplicação destes movimentos será igual ao valor anterior da solução somado ao valor dos movimentos.*

$$m_1 \parallel m_2 \implies f(m_1 \circ m_2 \circ s) = \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad (2.12)$$

Demonstração. Suponhamos que m_1 e m_2 sejam independentes mas $f(m_1 \circ m_2 \circ s) \neq \widehat{m}_1(s) + \widehat{m}_2(s) + f(s)$.

$$f(m_1 \circ m_2 \circ s) \neq \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad \text{De (2.5)}$$

$$\widehat{m}_1(m_2 \circ s) + f(m_2 \circ s) \neq \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad \text{De (2.5)}$$

$$\widehat{m}_1(m_2 \circ s) + \widehat{m}_2(s) + f(s) \neq \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad \text{Como } m_1 \parallel m_2$$

$$\widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \neq \widehat{m}_1(s) + \widehat{m}_2(s) + f(s) \quad \text{Logo uma contradição } \perp$$

Assim, por *reductio ad absurdum*, temos que se m_1 e m_2 são independentes então $f(m_1 \circ m_2 \circ s) = \widehat{m}_1(s) + \widehat{m}_2(s) + f(s)$. \square

2.4.2.4 Movimentos fortemente independentes – strongly independent moves

Dois movimentos m_1 e m_2 são fortemente independentes quando são independentes e podem ser aplicados simultaneamente a uma solução sem que a alteração feita por um gere conflito na causada pelo outro, a independência de movimentos pode ser definida para movimentos de vizinhanças diferentes. Formalmente temos que dois movimentos m_1 e m_2 são fortemente independentes $m_1 \parallel_e m_2$ se e somente se:

$$m_1 \parallel_e m_2 \iff m_1(m_2(s)) = m_2(m_1(s)) = m_1 \circ m_2 \circ s = m_2 \circ m_1 \circ s \quad (2.13)$$

Pela Equação 2.13 pode-se perceber que movimentos fortemente independentes são operações comutativas, pela própria definição.

Assim podemos estender a definição da vizinhança k para:

$$N^k(s) = \{m_i^k \circ s \mid \forall m_i^k \in M^k\} \quad (2.14)$$

Cabe aqui destacar que a independência de movimentos é uma relação dada dois a dois entre os movimentos, logo não existe transitividade na relação de independência destes, ou seja, se temos dois movimentos independentes $m_1 \parallel m_2$ e outro movimento m_3 tal que $m_2 \parallel m_3$ então **não** implica que $m_1 \parallel m_3$. Pode existir um conflito, logo os movimentos $m_1 \nparallel m_3$ ou seja, seriam conflitantes. Assim em termos de conflitos entre movimentos podemos escrever:

$$m_1 \parallel m_2 \wedge m_2 \parallel m_3 \not\Rightarrow m_1 \parallel m_3 \quad (2.15)$$

Demonstração. Consideremos o seguinte contraexemplo a seguir para a vizinhança de Swap, sendo os movimentos $Swap(2,3)$, $Swap(3,6)$, $Swap(4,5)$, neste caso podemos ver que $Swap(2,3) \parallel Swap(4,5)$ e que $Swap(4,5) \parallel Swap(3,6)$ contudo $Swap(2,3) \nparallel Swap(3,6)$. \square

Note, pela Figura 10, que $m_2 \circ m_3$ opera como um movimento 4-opt (quatro arcos são removidos e inseridos), assim a operação de composição do multi-improvement é capaz de lidar com movimentos de dimensões maiores, em uma única iteração.

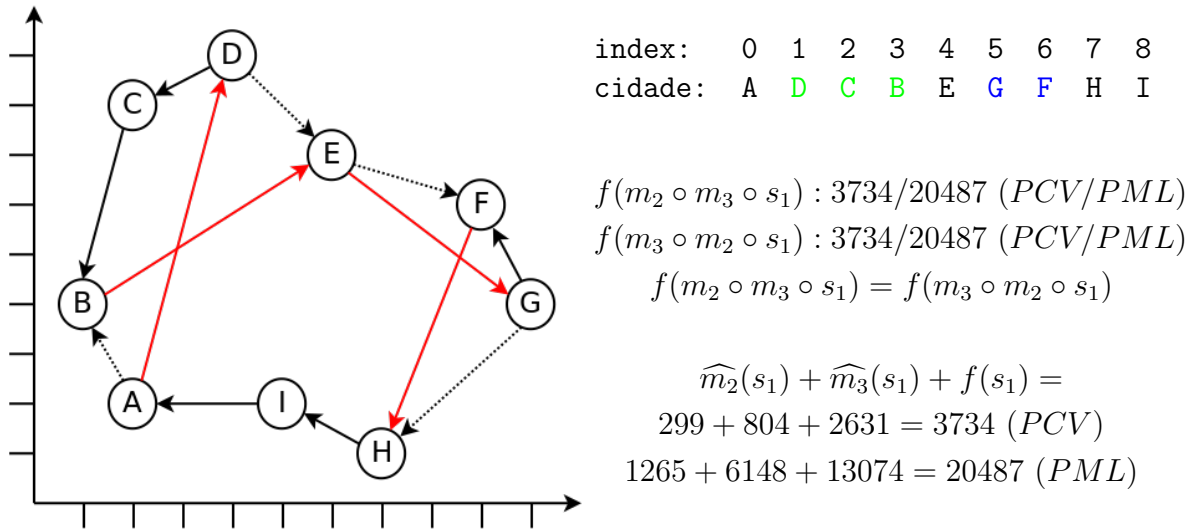


Figura 10 – Solução $m_2 \circ m_3 \circ s_1 = m_3 \circ m_2 \circ s_1$, com m_2 sendo 2-opt(5,6) e m_3 sendo 2-opt(1,3).

2.4.2.5 Movimentos conflitantes

Há movimentos com conflito de escrita na representação (estrutura de dados) da solução, portanto não é possível aplicá-los simultaneamente. Nesse caso, diz-se que eles são dependentes de ordem (order-dependent moves) ou, simplesmente, conflitantes.

Por exemplo, como $m_1=2\text{opt}(1,7)$ e $m_3=2\text{opt}(1,3)$ são conflitantes, $m_1 \circ m_3 \circ s_1$ e $m_3 \circ m_1 \circ s_1$ são soluções estruturalmente distintas, conforme pode ser visto na Figura 11.

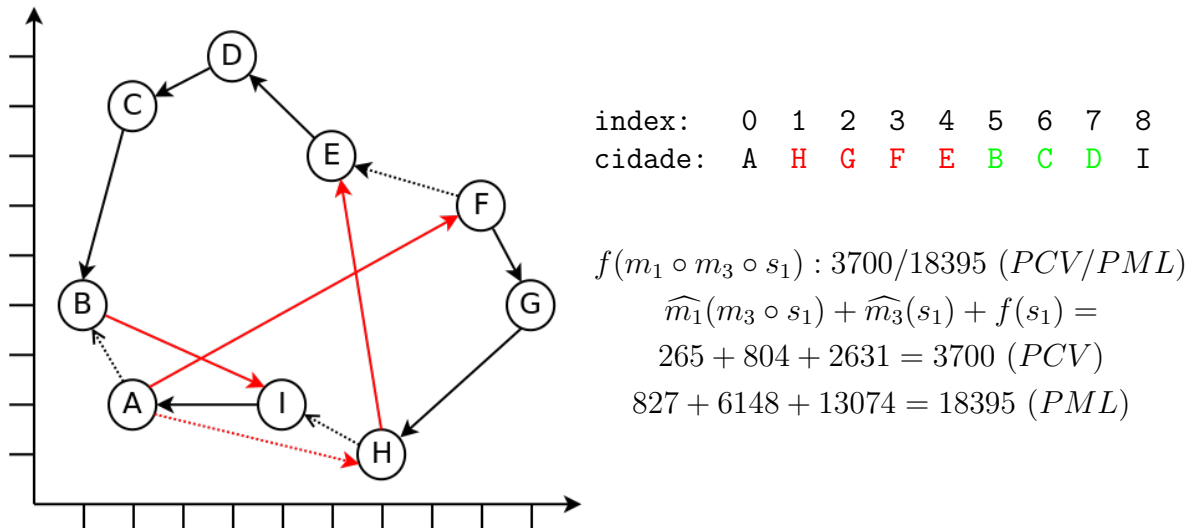


Figura 11 – Solução $m_1 \circ m_3 \circ s_1$, com m_1 sendo 2-opt(1,7) e m_3 sendo 2-opt(1,3).

Naturalmente, como m_1 e m_3 são conflitantes (vide Figura 12), $\widehat{m}_3(m_1 \circ s_1) \neq \widehat{m}_3(s_1)$ e $\widehat{m}_1(m_3 \circ s_1) \neq \widehat{m}_1(s_1)$. Também é interessante de notar que a composição executa um movimento 3-opt, então mesmo que não seja permitido na composição do 2-opt pode ser alcançado ao utilizar alguma outra vizinhança num processo simultâneo (generalized multi-improvement).

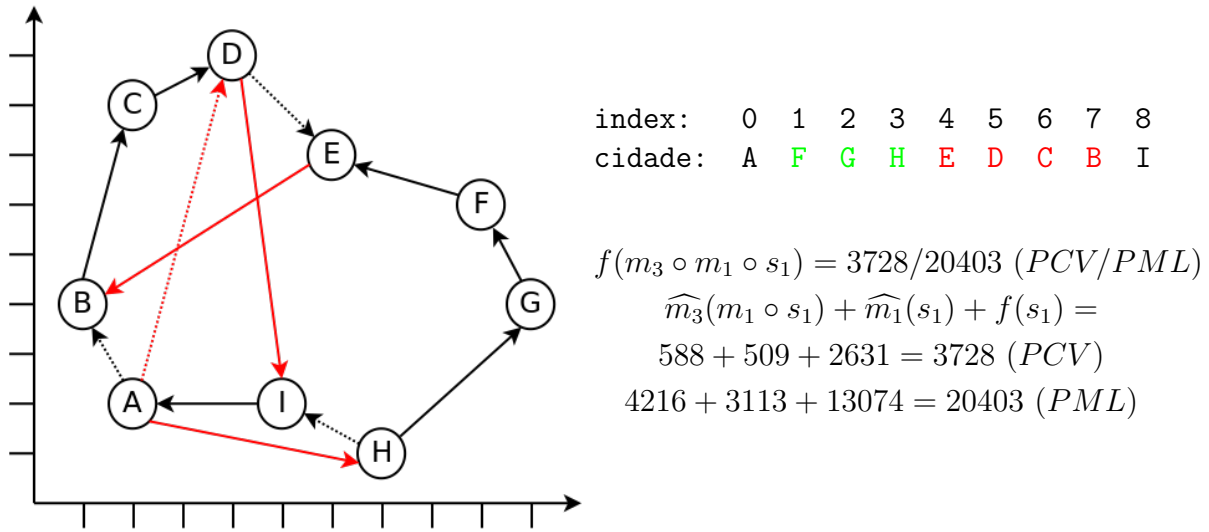


Figura 12 – Solução $m_3 \circ m_1 \circ s_1$, com m_3 sendo 2-opt(1,3) e m_1 sendo 2-opt(1,7).

2.4.3 First Improvement vs Best Improvement

As estratégias *First Improvement* (Primeira melhora) e *Best Improvement* (Melhor melhora) recebem como parâmetro a solução da iteração corrente para gerar seus vizinhos e escolhem uma solução a ser retornada conforme um critério específico, a saber, a primeira solução a melhorar a atual e a melhor solução encontrada na vizinhança, respectivamente.

Algoritmo 1 First Improvement para um problema de minimização

```

1: função FIRSTIMPROVEMENT(Solução:  $s$ , Operador de vizinhança:  $x$ )
2:   para  $m_i \in M$  faça                                ▷ Para cada movimento  $m_i \in M$ 
3:     se  $\widehat{m}_i(s) < 0$  então                                ▷ Se a solução for melhor que a atual
4:       retorna  $m_i \circ s$ 
5:   retorna  $s$                                 ▷ Caso não consiga melhorar retorna a própria solução

```

Podemos ver o pseudocódigo do *First Improvement* no Algoritmo 1 que consiste de enumerar os vizinhos até encontrar o primeiro que seja melhor que a solução atual, este então é retornado como resposta do método. O método de *Best Improvement* (Algoritmo 2) consiste em enumerar toda a vizinhança guardando a informação do melhor encontrado até o momento, e então retornar o melhor resultado encontrado.

Algoritmo 2 Best Improvement para um problema de minimização

```

1: função BESTIMPROVEMENT(Solução:  $s$ , Operador de vizinhança:  $x$ )
2:    $s^{best} \leftarrow s$                                 ▷ Melhor solução encontrada
3:   para  $m_i \in M$  faça                                ▷ Para cada movimento  $m_i \in M$ 
4:     se  $\widehat{m}_i(s) < f(s^{best}) - f(s)$  então                ▷ Se a solução for melhor que a atual altera a melhor solução encontrada
5:        $s^{best} \leftarrow m_i \circ s$ 
6:   retorna  $s^{best}$                                 ▷ Retorna a melhor solução encontrada

```

O *First Improvement* pode ser uma opção ao método de *Best Improvement* quando a enumeração de toda a vizinhança é uma atividade muito custosa. Embora não haja

um paralelo para a definição matemática formal da solução s' retornada pelo *First Improvement* esta pode ser definida para o *Best Improvement* de maneira simples por $s' \in N^x(s) \mid f(s') < f(s), \forall s \in N^x(s)$, o que, como veremos a seguir na seção 2.5, corresponde ao ótimo local para a solução s segundo a vizinhança x . Em termos de movimento temos $s' = m' \circ$ com $\widehat{m'}(s) < \widehat{m_i}(s) \mid \forall m_i \in M$.

2.4.4 Random Selection

Nesta estratégia *Random Selection* (Escolha Aleatória) é selecionada uma solução aleatoriamente entre aquelas que melhoram a solução atual.

Algoritmo 3 Random Selection para um problema de minimização

```

1: função RANDOMSELECTION(Solução:  $s$ , Operador de vizinhança:  $x$ )
2:    $S_{imp} \leftarrow \emptyset$  ▷ Conjunto com soluções de melhora
3:   para  $m_i \in M$  faça ▷ Para cada movimento  $m_i \in M$ 
4:     se  $\widehat{m_i}(s) < 0$  então ▷ Se a solução for melhor que a atual
5:        $S_{imp} \leftarrow S_{imp} \cup \{m_i \circ s\}$  ▷ Adiciona ao conjunto de soluções de melhora
6:   retorna  $Any(S_{imp})$  ▷ Retorna uma das soluções de melhora

```

A estratégia *Random Selection* (mostrada no Algoritmo 3) navega pelas soluções e na mantém as melhores soluções que melhoram a solução atual, conforme linha 5, para ao final retornam uma deste grupo.

2.4.5 Multi Improvement

Uma alternativa ao *Best Improvement*, *First Improvement* e ao *Random Selection* é o *Multi Improvement* (RIOS et al., 2015). A ideia é combinar um conjunto de movimentos independentes e executá-los simultaneamente sobre a solução de entrada. Note que, embora consista na aplicação de diversos movimentos, somente uma única solução vizinha é gerada. O *Multi Improvement* pode ser utilizado em qualquer contexto que o *Best Improvement* ou *First Improvement* se encaixe (etapa de Exploração de Vizinhança ou *Neighborhood Exploration*), porém caso só exista um único movimento independente na vizinhança, ele terá comportamento equivalente ao *Best Improvement*. Assim a solução s' retornada por uma iteração do *Multi Improvement* após ser aplicado a uma solução s é dada por $s' = m_1 \circ m_2 \circ \dots \circ m_k \circ s$ com os movimentos independentes $\{m_1, m_2, \dots, m_k\} \subset M$.

O *Multi Improvement* se encaixa particularmente bem com o conceito de *SIMD* (Single Instruction Multiple Data), presente nas GPUs, sendo sua complexidade similar ao *Best Improvement* (todos movimentos da vizinhança são enumerados), seguido de uma etapa de junção (ou *merge*) dos movimentos independentes. Podem existir cenários em que o *Best Improvement* seja mais eficiente (com poucos movimentos independentes), embora já tenha sido demonstrado na literatura que mesmo casos com apenas dois movimentos independentes acabam mais promissores no *Multi Improvement* do que no *Best Improvement*.

2.4.6 Passo iterativo

Em geral, um algoritmo de busca local é um processo iterativo pesado que tem como objetivo encontrar uma solução melhor que a atual dentro de um espaço de busca. A solução recebida como entrada pode ser aleatória ou advinda de alguma heurística construtiva, a intenção do processo é aprimorar o resultado encontrado.

Cada iteração da busca local tenta encontrar a melhor solução mediante alguma alteração na solução atual, então o processo se repete na solução gerada até que nenhuma melhora seja possível.

Algoritmo 4 Busca local definida de forma genérica

```

1: função LOCALSEARCH(Solução:  $s$ )
2:   enquanto  $f(\text{Alterar}(s)) < f(s)$  faça    ▷ Cada iteração corresponde a um passo
        iterativo da busca local
3:      $s \leftarrow \text{Alterar}(s)$ 
4:   retorna  $s$                                 ▷ Retorna a melhor solução encontrada

```

Supondo que $\text{Alterar}(s)$ (apresentado do Algoritmo 4) retorna uma solução melhor que a atual segundo alguma alteração, convençionemos então chamar de **passo iterativo** cada iteração da busca local em que o processo obtém uma solução melhor que a atual e salva o melhor resultado encontrado até o momento. Assim para uma solução s o passo iterativo é dado pela Equação 2.16.

$$\rho(s) = \min(s, \text{Alterar}(s)) \implies f(\rho(s)) \leq f(s) \quad (2.16)$$

2.5 Ótimo global vs Ótimo local

O ótimo global e ótimo local são exemplificados na Figura 1. Uma solução $s^* \in S$ é dita **ótimo global** para um problema Π quando não existe outra solução viável s' com melhor valor de função objetivo, formalmente temos que s^* é ótimo global quando:

- $\forall s' \in S \mid f(s') \leq f(s^*), s' \neq s^*$ para um problema de maximização;
- $\forall s' \in S \mid f(s') \geq f(s^*), s' \neq s^*$ para um problema de minimização.

Considere uma busca local de *Best Improvement* H para o problema Π sobre a estrutura de vizinhança N^x , após aplicar H a uma solução inicial $s^0 \in S$ é obtido um conjunto de soluções $N^x(s^0)$ vizinhas, assim o **ótimo local** (*mínimo local*) segundo a vizinhança N^x para a solução s^0 é dado por:

- $s'' \in N^x(s^0) \mid f(s'') < f(s'), \forall s' \in N^x(s^0), s'' \neq s'$, para um problema de minimização;
- $s'' \in N^x(s^0) \mid f(s'') > f(s'), \forall s' \in N^x(s^0), s'' \neq s'$, para um problema de maximização.

Em linhas gerais um **ótimo local** é a solução com melhor valor de função objetivo para um contexto local, seja uma vizinhança ou o conjunto imagem de uma heurística.

2.6 Meta-heurísticas

Uma meta-heurísticas diferencia-se de uma heurística por não ser acoplada a um problema específico ou classe de problemas. Meta-heurísticas fazem uso de artifícios capazes de encontrar soluções e aprimorar as já encontradas enquanto procuram escapar de mínimos locais. Na literatura são utilizadas em trabalhos que tratam de problemas da classe NP-Difícil devido a simplicidade de implementação e a intratabilidade da solução exata

de tais problemas. Assim, esses algoritmos são ferramentas robustas para serem aplicadas na resolução prática de problemas de otimização combinatória, sendo uma alternativa quando o respectivo algoritmo exato não é conhecido ou exige um alto tempo de execução (GLOVER; KOCHENBERGER, 2006).

As meta-heurísticas evoluem iterativamente conforme a parametrização fornecida até atingir um **Critério de Parada**, que pode também estar sujeito aos parâmetros do método. O critério de parada, em geral, está associado ao número de iterações, tempo de execução, um parâmetro de qualidade ou número de iterações sem melhora.

2.7 Dataflow

Atualmente os processadores no mercado de computadores seguem, em geral, o modelo de *Von Neumann*. No referido modelo, a execução das instruções é guiada por um fluxo de controle, ou seja, segundo a ordem que aparecem no programa, desta forma se faz necessário um *Program Counter* (Contador de Programa) para indicar qual a próxima instrução a ser executada. O contador também pode ser alterado por instruções de desvio, e laços de repetição ou qualquer tipo de comando de execução condicional.

Note que este modelo é intrinsecamente sequencial. No entanto, tenta-se resgatar paralelismo em nível de instruções com técnicas como pipelining (PATTERSON; HENNESSY, 2003), predição de desvio (PATTERSON; HENNESSY, 2003) e renomeamento de registradores (HENNESSY; PATTERSON, 2011).

O modelo dataflow (GURD; KIRKHAM; WATSON, 1985; SWANSON et al., 2003; DENNIS; MISUNAS, 1974; DAVIS, 1978; SAKAI et al., 1989; SHIMADA et al., 1986; KISHI; YASUHARA; KAWAMURA, 1983; GRAFE et al., 1989; PAPADOPOULOS; CULLER, 1990; SWANSON et al., 2007) expõe paralelismo de forma natural. Neste modelo, as instruções são executadas de acordo com o fluxo de dados, ou seja, assim que todos os seus operandos de entrada estiverem disponíveis.

No modelo dataflow os programas são escritos como um grafo de fluxo de dados onde os nós representam as instruções e as arestas direcionadas indicam as dependências de dados. Assim $A \rightarrow B$ indica que A produz um dado que é enviado como entrada para B após ter sido processado. Cabe lembrar que este modelo é adotado nas máquinas de Von Neumann para extrair paralelismo ao implementar o mecanismo de execução fora-de-ordem com escalonamento dinâmico baseado em fluxo de dados (TOMASULO, 1967), contudo limitado o paralelismo pela emissão das instruções que permanece seguindo o fluxo de controle. Numa arquitetura que segue totalmente o fluxo de dados as instruções não são emitidas segundo se apresentam no programa, instruções distintas podem executar concorrentemente.

Na Figura 13 pode ser visto um programa simples, à esquerda é mostrado o código e à direita sua tradução no grafo de fluxo de dados associado, note que as instruções de soma e multiplicação podem ser executadas em paralelo ou qualquer ordem sem alterar o resultado final.

2.7.1 Sucuri

Sucuri (ALVES et al., 2014) é uma biblioteca minimalística Dataflow para a linguagem Python que permite aos programadores explorarem o paralelismo mais naturalmente pela execução dataflow em máquinas que seguem a arquitetura Von Neumann. Sucuri permite uma execução transparente em um cluster de máquinas ao fazer uso do mecanismo de serialização de objetos em Python (Pickle).

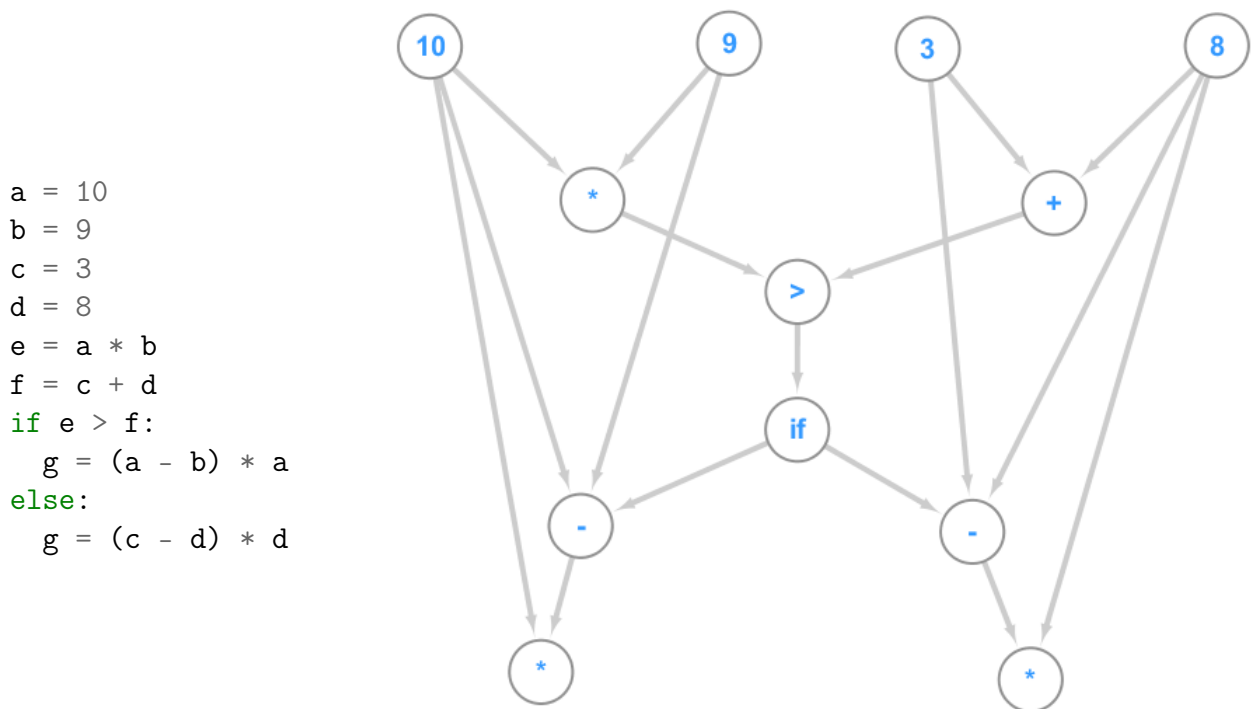


Figura 13 – Exemplo de conversão código para grafo de dependências. À esquerda pode ser visto um trecho de código em python ao passo que à direita é exibido o grafo dataflow associado.

Conceitualmente, um grafo dataflow é composto de nós que representa tarefas, que podem ser de grão fino (como instruções) ou grão grosso (como funções ou procedimentos). As arestas que conectam os nós representam dependências de dados, significando que um nó de origem irá produzir um resultado que será utilizado por um nó de destino. Quando um certo nó recebe todas as entradas necessárias (são satisfeitas todas as suas dependências) este pode ser enfileirado para executar o seu processamento.

A Figura 14 mostra a estrutura do Sucuri, onde é possível observar três componentes principais: **Graph** (Grafo), **Scheduler** (Escalonador) e o **Worker** (Operário).

O **Graph** é apenas o grafo de dependências da aplicação dataflow, podendo ser visto como um contêiner de nós, onde cada um contém:

- A lista de entradas de dados até o momento. Quando todos os operandos são recebidos ocorre um matching e a execução do nó será disparada;
- A função (computação) que deve ser executada quando for o momento para este nó;
- Uma lista de nós de destino que devem receber o resultado produzido na sua computação;
- Atributos específicos, como um identificador único que pode ser usado para atribuição de trabalho para um conjunto específico de nós (como numa abordagem fork-join).

Quando é usado em um cluster de computadores, cada componente supracitado é replicado em cada máquina do cluster, com exceção do **Scheduler**, o que implica que a Sucuri adota um pool de tarefas centralizado. Em (SILVA et al., 2016) os autores da

Sucuri implementaram e avaliaram uma versão da biblioteca utilizando um escalonador distribuído, contudo esta versão ainda não foi utilizada nesse trabalho.

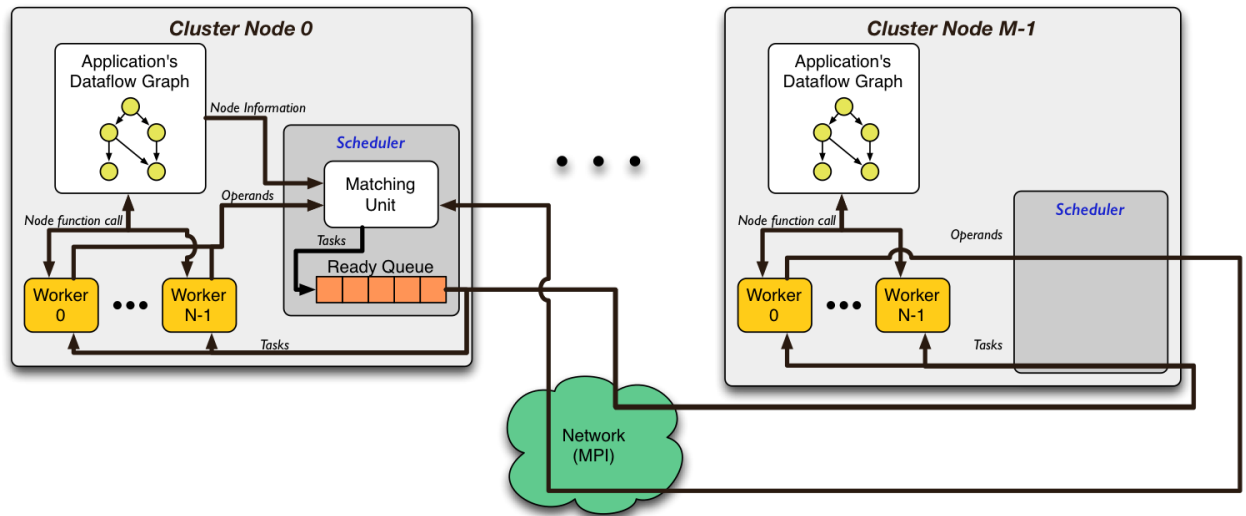


Figura 14 – A Arquitetura da Sucuri (de (ALVES et al., 2014)). A mesma estrutura é replicada em cada nó, entretanto apenas no Scheduler do nó 0 do cluster contem o Matching Unit e o Ready Queue. Este é responsável por receber operandos dos workers locais e de outros Schedulers, além disso este gera as tarefas para serem enfileiradas no Ready Queue.

O Scheduler (escalonador) principal, situado no nó 0 do cluster, é composto por uma Matching Unit, uma Ready Queue (fila de prontos) e uma Waiting Queue (fila de espera). Este é responsável por entregar todos os operandos aos seus respectivos nós de destino no grafo. Se ocorre um match, isto é, todas as dependências de dados são satisfeitas para um nó, então uma tarefa é criada e inserida na Ready Queue. Quando os workers estiverem ociosos estes vão requisitar uma tarefa para o Scheduler que será buscada da Ready Queue. O Scheduler situado em outros nós do cluster são mais simples e apenas encaminham tarefas do Scheduler principal para seus workers locais e operandos de seus workers para o Scheduler principal. O grafo é replicado em todos os nós do cluster mas apenas o grafo do nó 0 pode receber operandos do Scheduler principal.

Toda a comunicação intra-nó entre os componentes principais citados acima é realizada via memória compartilhada e entre Schedulers de nós diferentes é feita via interface.

Cada nó do grafo da Sucuri é associado a uma função que pode ser implementada pelos programadores e passada para estes no momento da criação do grafo dataflow. Após instanciar os nós, o programador pode criar as ligações entre estes usando o método `add_edge()` do nó que cria uma dependência no grafo dataflow. Quando o escalonador dispara a execução de uma tarefa num certo worker, este vai chamar a execução do método `run()` do respectivo nó para aquela tarefa. Na maioria dos casos, este método `run()` vai atuar como um invólucro que chama a execução da função associada ao nó no momento da construção do grafo, passando os operandos como parâmetro e ao final irá enviar os valores retornados pela função para o escalonador.

De mais a mais Sucuri também provê um conjunto especial de nós que podem auxiliar o programador a elaborar aplicações que sigam alguns padrões de paralelismo. Por exemplo, uma aplicação que envolva pipeline para Sucuri é apresentada na Figura 15. O painel A mostra o grafo representando este padrão e o painel B o código Sucuri para essa operação.

Note como novos nós são criados (linhas 11-13), adicionados ao grafo (linhas 17-19) e como as arestas conectando os nós são definidas (linhas 21 e 22). Perceba também a instanciação no escalonador (linha 6) e como este é inicializado após o grafo dataflow ter sido definido (linha 23).

A Figura 15 também aponta como o nó especial **Source** recebe um objeto iterador Python (por exemplo uma lista ou um descritor de arquivo) na sua instanciação. Durante a execução do programa, o método `run()` do nó **Source** será disparado apenas uma vez visto que este é a raiz, i.e., não possui operandos de entrada no grafo e é usado para iniciar a computação. Todavia, a execução deste método vai durar até que todo o conteúdo do objeto iterador tenha sido consumido. Por padrão o nó **Source** vai executar um laço sobre o objeto iterador e produzirá múltiplas saídas (mensagens) que irão disparar a execução do pipeline múltiplas vezes.

Veja também que o último nó do pipeline é o nó especial **Serializer**, que é responsável por escrever os dados no arquivo. É possível que os dados produzidos pelo nó **Source** sejam processados fora de ordem pelo segundo nó uma vez que múltiplas tarefas podem ser escalonadas em workers diferentes. Por conseguinte, é necessário reordenar os dados antes de os escrever no arquivo, no nó **Serializer**. Para tanto, os dados produzidos pelo nó **Source** precisam estar encapsulados num objeto **TaggedValue** que contem uma atributo `tag`, indicando sua posição na ordem do conjunto de dados. O nó no intermédio também vai enviar os dados filtrados dentro do objeto **TaggedValue**, com a mesma tag do pedaço de dados que recebeu. O nó **Serializer** então ao receber os dados do nó de filtro, vai armazená-los num buffer ordenando de acordo com a tag. Se a tag do último fragmento de dados recebido corresponde ao próximo a ser escrito no arquivo então o nó **Serializer** obtém os dados do buffer ordenando e escreve no arquivo até que haja uma lacuna na ordenação, i.e. a porção de dados que é a próxima a ser escrita ainda não chegou ao buffer. Se os dados recebidos pelo **Serializer** estão fora de ordem, o nó apenas os armazena no buffer ordenado e aguarda por mais dados. O método `pin` é usado para fixar um nó a um determinado worker, o que fará com que este seja executado apenas por este worker específico. No caso do exemplo, foram fixados os nós que fazem operações de ES no disco aos workers que tem acesso direto ao disco.

Um recurso de modelagem interessante que foi usado nesse trabalho consiste em explorar a programação em grão grosso usando a Sucuri junto com a computação em grão fino em GPU, resultando assim numa computação heterogênea Dataflow/Von Neumann, com nós dataflow executando operações de alta performance em GPU. Esta estratégia permite uma grande flexibilidade para o design de insólitos algoritmos, com maior simplicidade do que adotando um único paradigma de computação (dataflow ou Von Neumann).

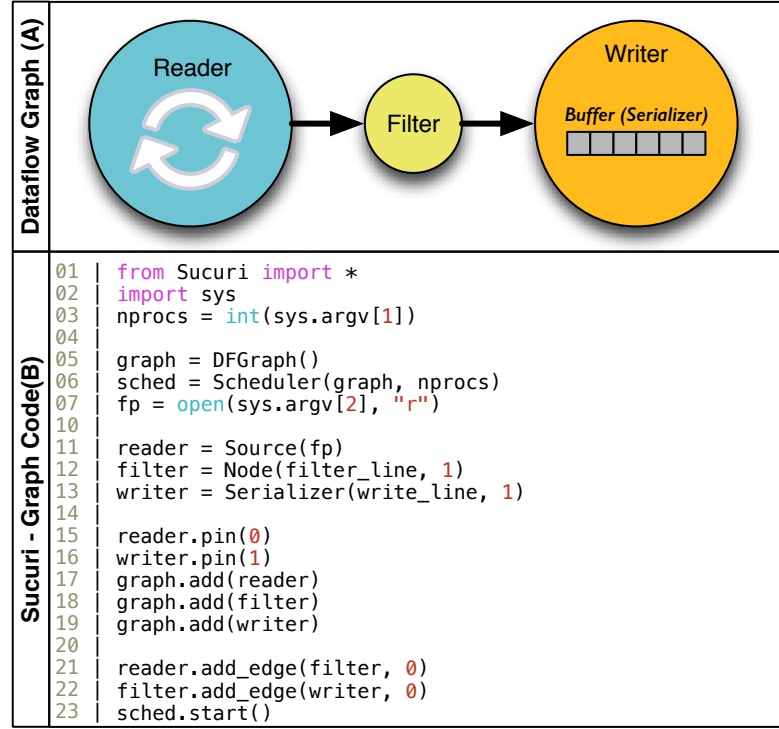


Figura 15 – Pipelining com Sucuri. Painel A mostra um grafo de uma aplicação dataflow, o painel B descreve o grafo usando a Sucuri.

2.8 RVND

O método RVND clássico (SOUZA et al., 2010) é apresentado no Algoritmo 5, como um procedimento sequencial, este depende do passo anterior para continuar o seu processamento, de acordo com o resultado anterior o método pode decidir se move para a próxima vizinhança ou volta para a primeira. Podemos ver no condicional da linha 5 que após obter a melhor solução para a vizinhança atual, o RVND verifica se esta é melhor que a solução atual, caso seja então o método volta pra a primeira vizinhança, caso contrário segue para a seguinte.

Algoritmo 5 RVND clássico

```

1: função RVND(Solução:  $s$ )
2:    $N \leftarrow \{N^1, \dots, N^{kmax}\}$  ▷ Vizinhanças em ordem aleatória
3:   para  $k \leftarrow 1$  to  $kmax$  faça
4:      $s' \leftarrow s''$  com  $f(s'') \leq f(s''') \mid \forall s''' \in N^k(s)$  ▷ Melhor solução de  $N^k(s)$ 
5:     se  $f(s') < f(s)$  então
6:        $s \leftarrow s'$ 
7:        $k \leftarrow 1$ 
8:     senão
9:        $k \leftarrow k + 1$ 
10:  retorna  $s'$ 

```

A cada iteração o RVND retorna uma solução $s' = m_y^x$ com $\widehat{m}_y^x(s) < \widehat{m}_i^x(s) \mid \forall m_i^x \in M^x$, que corresponde à melhor solução para a vizinhança atual.

Considerando $\mathcal{M} = M^{RVND} = M^1 \cup M^2 \cup \dots \cup M^k$ o conjunto com os movimentos

de todas as vizinhanças usadas pelo RVND, então em termos de movimento temos que a solução s'' retornada ao final do RVND pode ser escrita como $s'' = m_z \circ s$ com $m_z \in \mathcal{M}$ e $\widehat{m}_z(s) < \widehat{m}_i(s) \mid \forall m_i \in \mathcal{M}$.

2.9 DVND

O *Distributed Variable Neighborhood Descent* DVND concebido por (RIOS et al., 2018) utiliza múltiplas vizinhanças conforme o faz o VND (*Variable Neighborhood Descent* proposto por (MLADENOVÍČ; HANSEN, 1997)) contudo propõe o processamento das vizinhanças de forma distribuída. Este processamento distribuído se dá pelo escalonamento das tarefas de enumeração das vizinhanças o que naturalmente proporciona a aleatoriedade proposta no RVND.

A implementação em dataflow não pode alcançar uma grande melhoria do RVND em termos de tempo ou qualidade da solução pois o grafo se assemelha a uma cascata (veja a Figura 16) o que não permite alcançar paralelismo, então se torna natural o uso do método DVND conforme o Algoritmo 6. A ideia do DVND é que quando uma solução atinge um ótimo local para uma estrutura de vizinhança ainda pode existir um vizinho com melhor valor de função objetivo em uma estrutura de vizinhança diferente, destarte não necessariamente sendo um ótimo local para todas as vizinhanças. Se uma melhoria é encontrada o processo de busca é reiniciado para todas as estratégias de vizinhança.

Algoritmo 6 DVND clássico

```

1: função DVND(Solução:  $s$ , Vizinhanças:  $N$ )
2:    $W \leftarrow \emptyset$ 
3:    $H \leftarrow \emptyset$ 
4:   para todo  $N_k \in N$  faça
5:      $s_k \leftarrow s$  ▷ Solução atual para vizinhança  $k$ 
6:      $H_{k,s} \leftarrow true$  ▷ Solução já foi enumerada pela vizinhança
7:      $W_k \leftarrow false$  ▷ Vizinhança aguardando solução
8:     Chame de forma assíncrona  $N^k(s_k)$ 
9:     enquanto  $\exists w \in W \mid w = false$  faça
10:       $k \leftarrow \text{join } N^k(s_k)$  ▷ Aguarda a resposta da vizinhança  $N^k$ 
11:       $s_k \leftarrow \text{Melhor solução de } N^k(s_k)$ 
12:      se  $f(s_k) < f(s)$  então
13:         $s \leftarrow s_k$ 
14:         $W_k \leftarrow true$ 
15:        para todo  $N_k \in N$  faça
16:          se  $W_k \wedge \neg H_{k,s}$  então
17:             $s_k \leftarrow s$ 
18:             $H_{k,s} \leftarrow true$ 
19:             $W_k \leftarrow false$ 
20:            Chame de forma assíncrona  $N^k(s_k)$ 
21:  retorna  $s$ 

```

Considerando $\mathcal{M} = M^{DVND} = M^1 \cup M^2 \cup \dots \cup M^k$ o conjunto com os movimentos de todas as vizinhanças usadas pelo DVND, então em termos de movimento temos que a solução s'' retornada a cada iteração do DVND pode ser escrita como $s'' = m_z \circ s$ com $m_z \in \mathcal{M}$ e $\widehat{m}_z(s) < \widehat{m}_i(s) \mid \forall m_i \in \mathcal{M}$. Vale ressaltar que $\mathcal{M} = M^{DVND} = M^{RVND}$,

a diferença dos métodos é que a cada iteração o RVND move para a melhor solução da vizinhança atual e no caso do DVND este move para a melhor solução entre todas as vizinhanças.

Numa análise em mais alto nível do RVND (Algoritmo 5) e DVND (Algoritmo 6), pensando-se à luz das estratégias de *First improvement* e *Best improvement*, o RVND enumera as soluções vizinhas da solução atual vizinhança por vizinhança até encontrar uma solução que a melhore e então retorna para a primeira vizinhança ao passo que o DVND enumera todas as vizinhanças para então optar pela solução de melhor valor. Desta forma o RVND é uma estratégia de *first improvement* no contexto de vizinhanças de solução e o DVND uma estratégia de *best improvement*.

3 METODOLOGIA PROPOSTA

Existem diversas oportunidades para desenvolvimento de novos algoritmos de otimização e busca local, utilizando recursos de programação paralela. Neste trabalho é proposta uma implementação em dataflow do DVND e apresentado GDVND conforme se verá a seguir.

3.1 RVND

O RVND, conforme descrito na Seção 2.8, pode ser implementado num grafo dataflow conforme a Figura 16, onde o nó inicial (*ini*) envia a solução inicial para o primeiro operador de vizinhança (*Swap*), cada nó operador de vizinhança explora todos os movimentos e escolhe o melhor, se este melhora a solução atual então a solução melhorada é enviada para o primeiro nó operador (*oper0*) caso contrário a solução é enviada para o próximo nó de enumeração de vizinhança.

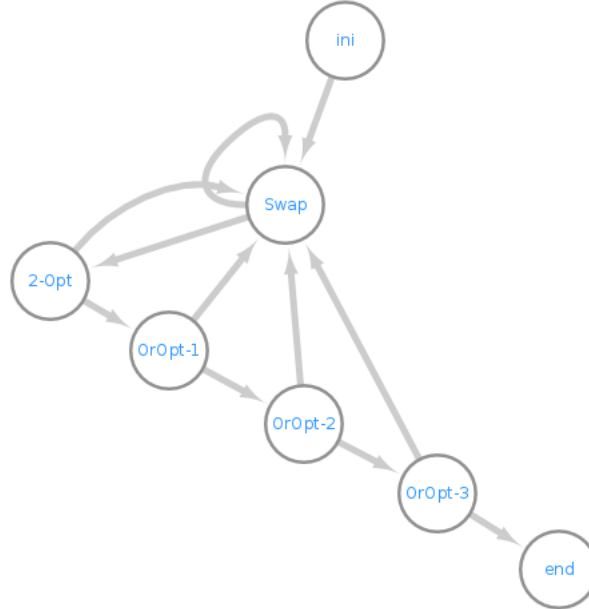


Figura 16 – Arquitetura simplificada do dataflow para o RVND com as vizinhanças utilizadas.

A implementação do nó operador (*Swap*, *2-Opt*, *OrOpt-1*, *OrOpt-2*, *OrOpt-3*) é tão simples quanto o Algoritmo 7 e cada estratégia de vizinhança é atrelada a um nó operador, sendo que a ordem desses é variada para cada execução do método para caracterizar o RVND, a decisão de para qual nó enviar o resultado é tomada pela configuração do dataflow. Quando a solução atinge o nó final (*end*) esta é salva e o processo termina.

Algoritmo 7 Nó de vizinhança do RVND

```

1: função RVND_OPER(Solução:  $s$ )
2:    $s' \leftarrow$  melhor solução de  $N^k(s)$ 
3:    $improvFlag \leftarrow f(s') < f(s)$ 
4:   retorna ( $s', improvFlag$ )
  
```

Pode ser visto destacado na Figura 17 uma vizinhança com suas ligações ao grafo dataflow, uma de entrada de dados e duas outras de saída que são para o caso de haver ou não uma melhoria no valor da solução. Para acoplar uma nova vizinhança ao algoritmo basta que seja inserido um novo nó de enumeração com sua entrada de dados vindo do nó anterior e duas saídas de dados, uma retornando para a primeira vizinhança e outra para a vizinhança seguinte, conforme destacado na mesma figura.

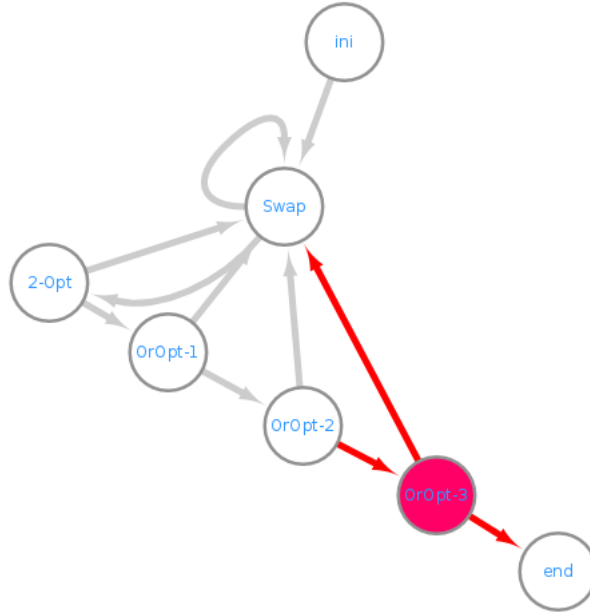


Figura 17 – Uma vizinhança e suas ligações ao grafo dataflow no RVND.

3.1.1 Passo iterativo

Utilizando o termo convencionado na seção 2.4.6, cada passo iterativo do RVND retorna a melhor solução encontrada para a vizinhança atual. Assim sendo N^k a vizinhança atual temos o passo iterativo para o RVND expresso na Equação 3.1.

$$\rho^{RVND}(s) = s' \in N^k \quad \text{sendo} \quad f(s') < f(s''), \forall s'' \in N^k(s) \wedge s'' \neq s \quad (3.1)$$

Fazendo uso da notação de movimentos temos podemos escrever 3.1 como a Equação 3.2.

$$\rho^{RVND}(s) = m \circ s \quad \text{com} \quad m \in M^k \wedge \widehat{m} < \widehat{m}_i \mid \forall m_i \in M^k \quad (3.2)$$

3.2 DVND

Foi descrito na Seção 2.9 o algoritmo do DVND, contudo um modelo dataflow para o DVND pode ser visto na Figura18, o método usa $P+1$ nós iniciais ($ini0, ini1, \dots iniP$) que

alimentam os nós operadores (*Swap*, *2-Opt*, *OrOpt-1*, *OrOpt-2*, *OrOpt-3*) e o nó gerente (*man*) com a solução inicial para a busca local.

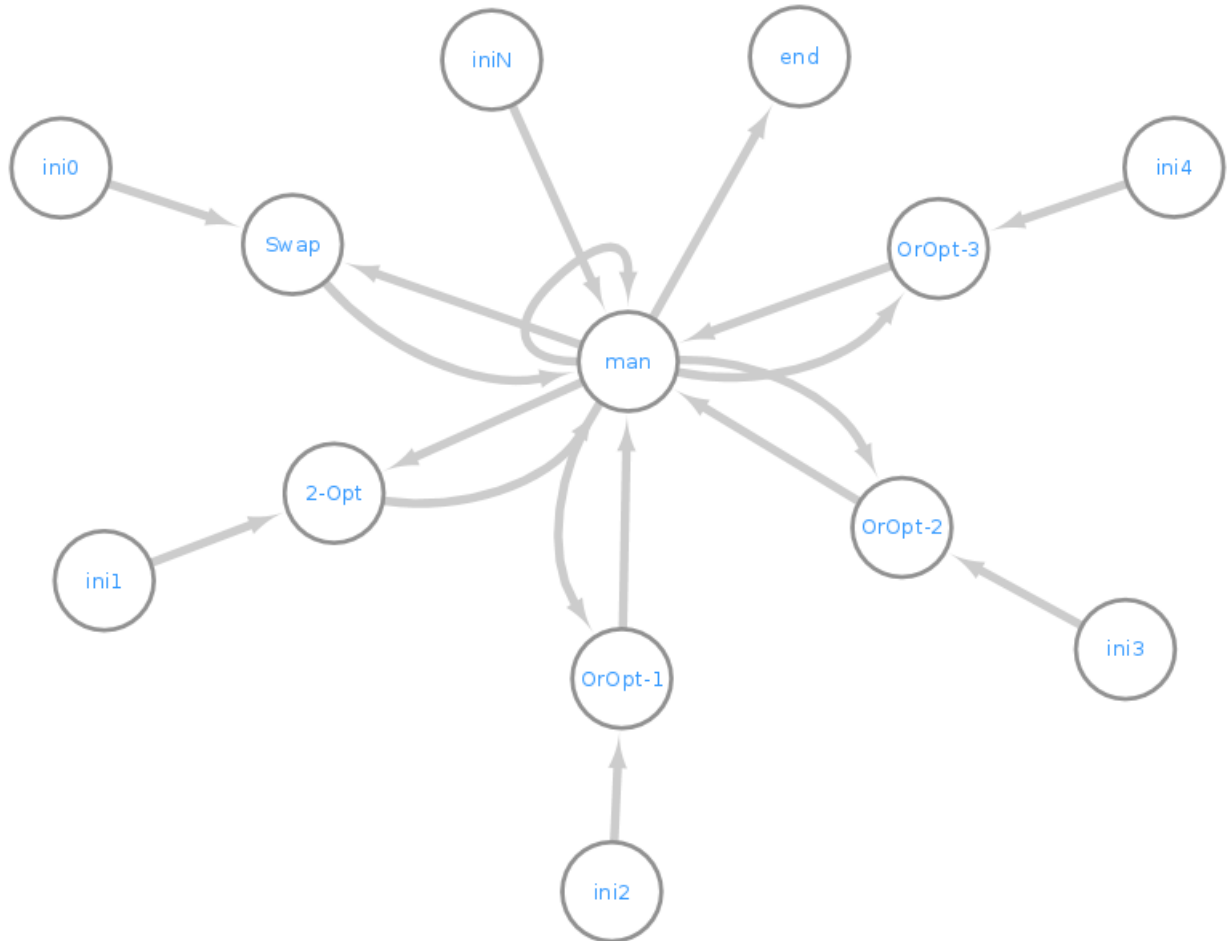


Figura 18 – Arquitetura simplificada do dataflow para o DVND com as vizinhanças utilizadas.

Cada nó operador utiliza uma estratégia de vizinhança diferente, quando termina de enumerar as soluções este retorna a melhor para o nó gerente que mantém a melhor solução encontrada até o momento. Então o nó gerente identifica a melhor solução conhecida e a envia de volta para o dataflow (como no Algoritmo. 8) que encaminha a solução para todos os nós operadores parados, o processo se repete até que nenhum nó operador encontre uma solução melhor. Então a melhor solução encontrada é enviada para o nó final (*end*) que salva o resultado da busca local.

Algoritmo 8 Nó *man* do DVND

- 1: **função** DVND_MAN(Solução: *s*, Histórico: *H*)
 - 2: $H \leftarrow H \cup \{s\}$
 - 3: **retorna** *bestSolution*(*H*)
-

O critério de parada é alcançado quando nenhum nó operador consegue melhorar a solução atual significando que está correspondendo a ótimo local para todas as vizinhanças

usadas no processo. Este é, de certa forma, o mesmo critério utilizado por ambos RVND e DVND, contudo o caminho percorrido pelos diferentes processos pode levar a ótimos locais diferentes. Tanto para o RVND quanto DVND as estratégias de vizinhança são atribuídas aos nós *oper* e os métodos não definem o comportamento interno destes nós facilitando sua alteração ou mesmo a adição de uma nova vizinhança.

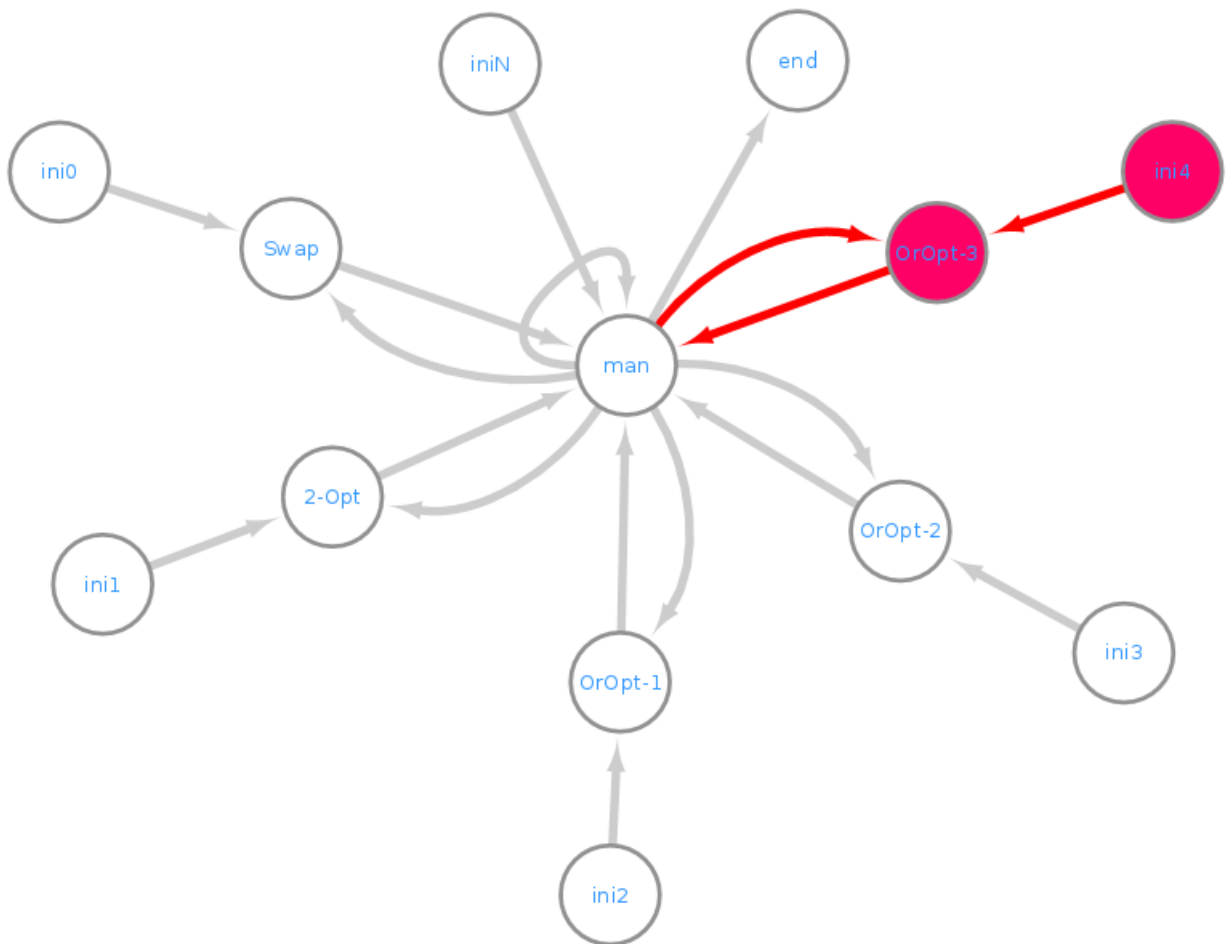


Figura 19 – Uma vizinhança e suas ligações ao grafo dataflow no DVND.

Como dito anteriormente, qualquer vizinhança pode ser facilmente acoplada ao procedimento apenas adicionando um novo nó operador *oper* ao grafo dataflow, conforme destacado na Figura 19, o que é uma grande característica do método, e como os nós são independentes é possível fazê-lo sem impactar nos outros nós. Inicialmente pode parecer que muitos nós operadores podem sobrecarregar o nó gerente, mas é importante notar que o tempo computacional gasto pelos nós operadores é geralmente muito mais caro (exploração das vizinhanças é o processo mais caro de uma meta-heurística), desta forma não será um grande problema mesmo quando os nós operadores responderem em tempos curtos.

No trabalho apresentado em (ARAÚJO; COELHO; MARZULO, 2018) o modelo dataflow usado possuía apenas uma porta de saída então a mensagem seria enviada para todos os nós a ele vinculados, numa melhoria a esta implementação foi implementado e sugerido um aperfeiçoamento ao Sucuri para comportar portas de saída diferentes para destinos diferentes. No modelo proposto (veja Figura 18), o nó gerente é ligado a todos os nós operadores o que causaria uma inundação (flooding) na rede, desta forma possuir mais

de uma porta de saída permite evitar que um nó processe indevidamente uma solução ou que receba uma mensagem com uma indicação de que não deve ser processada o que causaria uma troca de mensagens desnecessária, problema esse que se intensifica ao rodar o processamento em rede. Um problema que permanece é o escalonador centralizado, o que significa que num processamento em rede cada mensagem precisa ser enviada de volta para o computador rodando o nó gerente para só então ser enviada para o seu nó de destino, incluindo nisso as mensagens de feedback explicadas no Sub-seção 3.2.2. Contudo a disponibilização de um escalonador distribuído demandaria um novo projeto para a o dataflow do DVND (Figura 18) de forma a fazer melhor uso deste recurso. Apesar das limitações atuais deste projeto em desenvolvimento, o Sucuri já provê um ambiente que pode ser facilmente escalado de uma máquina para um experimento em rede completamente distribuído.

3.2.1 Passo iterativo

Utilizando o termo convencionado na seção 2.4.6, cada passo iterativo do DVND retorna a melhor solução encontrada para todas as vizinhanças. Dessa forma, sendo N a união de todas as vizinhanças, o passo iterativo do DVND pode ser dado pela Equação 3.3.

$$\rho^{DVND}(s) = s' \in N \quad \text{com} \quad f(s') < f(s''), \forall s'' \in N(s) \wedge s'' \neq s \quad (3.3)$$

Fazendo uso da notação de movimentos podemos escrever 3.3 como a Equação 3.4.

$$\rho^{DVND}(s) = m \circ s \quad \text{com} \quad m \in \mathcal{M} \wedge \widehat{m} < \widehat{m}_i \mid \forall m_i \in \mathcal{M} \quad (3.4)$$

Pelo passo iterativo do RVND ρ^{RVND} (3.2) e do DVND ρ^{DVND} (3.4) podemos concluir que $\rho^{DVND}(s) \leq \rho^{RVND}(s)$, contudo isso não é suficiente para afirmar que o DVND encontre necessariamente melhores resultados.

3.2.2 Nó de flip flop

É esperado um comportamento sem estado para os nós num grafo de uma arquitetura dataflow contudo no modelo do DVND proposto (veja Figura 18) o nó gerente precisa manter o histórico da melhor solução já encontrada até o momento e decidir se o método alcançou ou não um ótimo local. Para alcançar este comportamento, é apresentado em (ARAÚJO et al., 2018) um nó de flip flop (nó *FF* na Figura 20), este nó possui duas portas de entrada, uma que de fato recebe a informação do nó anterior e outra que o retro-alimenta com sua própria saída para manter o histórico do resultado. A intenção desse padrão é conceder ao nó a capacidade de realizar uma decisão baseada na última iteração sem necessariamente tornar o nó statefull.

É importante ressaltar que o nó *oper0* no RVND (Figura 16) também possui uma retroalimentação contudo não é um nó de flip flop pois não retroalimenta sua própria saída, este nó não precisa da informação de sua última execução para realizar o seu processamento, desta forma possui apenas uma porta de entrada.

3.2.3 Múltiplas portas de saída

Em modelo dataflow um nó será processado quando todas as suas dependências forem satisfeitas, contudo, ao final de seu processamento o resultado pode ser enviado como entrada para um ou mais nós. Pode ser visto na Figura 21 que o nó *MO* está conectado

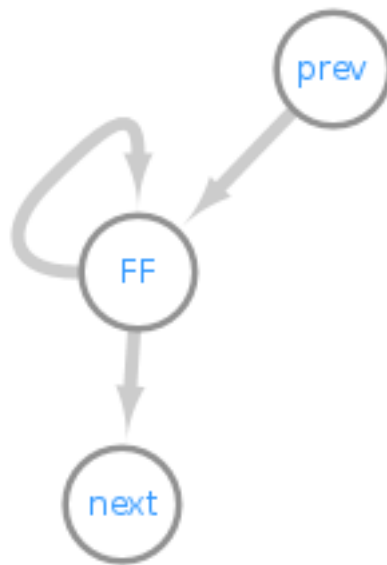


Figura 20 – FF identifica o nó de flip flop.

a um nó com informação de entrada (nó *prev*) e ligado a três nós de saída (*next 0*, *next 1* e *next 2*).

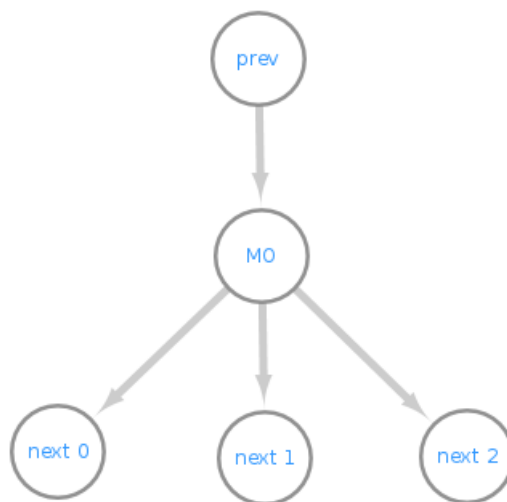


Figura 21 – Pedaço de um grafo dataflow em que o nó *MO* do dataflow possui múltiplas saídas.

No modelo atual de implementação da biblioteca Sucuri é possível que um nó receba informações de vários outros nós contudo existe apenas uma porta de saída a qual podem estar conectados vários nós.

Pegando-se o exemplo da Figura 22, quando o nó *MO* termina de processar seu resultado é enviado para a única porta de saída existente a qual estão ligados os nós *next 0*, *next 1* e *next 2*, dessa forma todos os nós seguintes vão receber a informação e estarão aptos para processamento mesmo que a informação não seja destinada a eles. Cada um desses nós terá então que verificar se a mensagem é destinada a ele antes de processar ou não a informação.

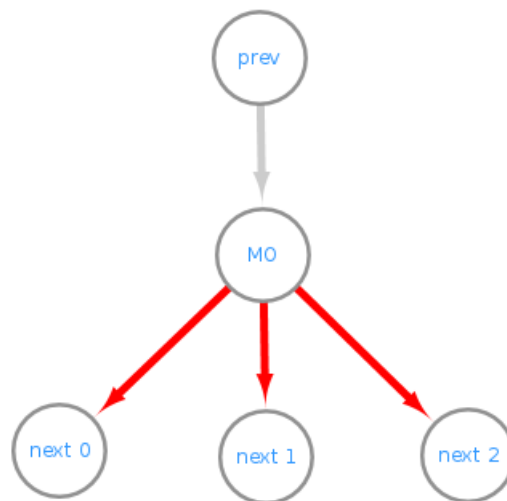


Figura 22 – Quando o nó *MO* termina de processar seu resultado é enviado para todos os nós subsequentes (*next 0*, *next 1* e *next 2*).

Foi implementada a alteração exemplificada na Figura 23, nesse caso existe uma porta de saída para cada nó ligado ao nó *MO*, este identifica os destinatários da mensagem e aciona apenas aqueles necessários.

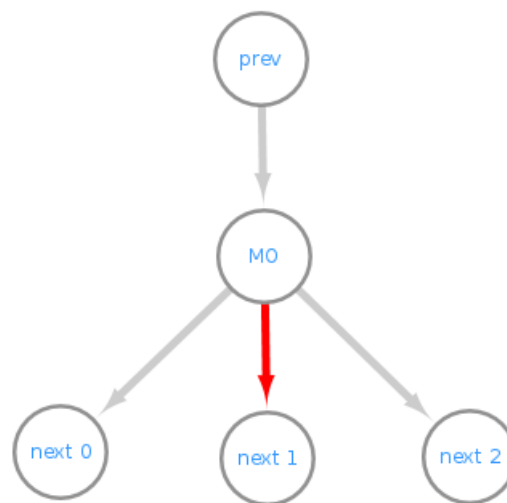


Figura 23 – Quando o nó *MO* termina de processar é possível escolher qual porta de saída será utilizada e assim decidir o destino da informação.

No DVND, ao escolher o destinatário da mensagem é possível evitar uma grande quantidade de transmissões desnecessárias de mensagens, evitando assim que os nós de vizinhança sejam acionados apenas para identificar que não precisam processar a mensagem. Vejamos, por exemplo, o nó *man* da Figura 18 sem o artifício de múltiplas portas de saída todas as mensagens serão enviadas para todas as 5 vizinhanças e para o nó finalizador (*end*) sendo que a mensagem só é enviada para os nós ociosos naquele momento.

A implementação de múltiplas portas de saída trás vantagens ainda mais significativas quando o tamanho da mensagem a ser transmitido é grande e quando a execução do processamento está sendo feita em um ambiente de mais de uma máquina o que envolve

troca de mensagens pela rede.

3.3 GDVND

Trazendo o conceito de *movimentos independentes* (discutido na Seção 2.4.2.3) para o DVND surge a ideia do Generalized Distributed Variable Neighborhood Descent (GDVND). A ideia do GDVND é um DVND que trabalha no escopo de movimentos de melhora e não em melhores soluções. No DVND, a cada iteração o método recebe uma nova solução de uma das vizinhanças e verifica qual a melhor para então enviar esta para ser processada pelas vizinhanças ociosas que ainda não a processaram, sempre mantendo a melhor solução já encontrada. A diferença para o GDVND é que este recebe não somente uma solução mas a solução com um conjunto de movimentos de melhora encontrado pela vizinhança, então o método tenta combinar os movimentos da melhor solução atual com aqueles da solução recebida. Para tanto é necessário que as soluções tenham a mesma solução base, então é necessário identificar o conjunto de movimentos independentes que proporcione o maior ganho em termos de valor da solução. Além de melhorar a solução a utilização da composição de movimentos proporciona uma melhor utilização dos recursos usados no processamento ao aproveitar computações realizadas em nós diferentes.

O grafo dataflow do GDVND é equivalente ao do DVND expresso na Figura 18 alterando apenas a informação trafegada entre os nós e a implementação interna do nó gerenciador, conforme Algoritmo 9, ao qual caberá combinar os movimentos de vizinhanças diferentes. Na linha 2 é chamado o processo combinar movimentos que pode ser visto em mais detalhes no Algoritmo 10. Os movimentos são combinados para formar uma solução resultante, tomando proveito do Teorema da independência de movimentos dois a dois (Teorema 3) o valor da solução resultante pode ser calculado para dois movimentos, sem a necessidade de recalculer toda a solução, o que é particularmente útil para grandes instâncias e problemas cujo cálculo da função objetivo é muito caro computacionalmente. Pode-se utilizar o mesmo teorema para estimar o valor das soluções na composição de mais de dois movimentos dois a dois.

Algoritmo 9 Nó *man* do GDVND

- 1: **função** GDVND_MAN(Solução: s , Histórico: H , Vizinhança: k)
 - 2: $s_{merged} \leftarrow GDVND_merge(s, bestSolution(H))$ \triangleright Combina a solução atual com a melhor conhecida
 - 3: $H(k) \leftarrow \min(s, s_{merged}, bestSolution(H))$ \triangleright Atualiza o histórico da vizinhança k com a melhor solução
 - 4: **retorna** SoluçãoBase($H(k)$), Movimentos($H(k)$)
-

Seja um grafo $G = (V, E)$ cujos vértices V são movimentos e as arestas E indicam se um movimento é independente de outro. Dessa forma encontrar o melhor sub-conjunto de movimentos independentes entre si dois a dois corresponderia a encontrar o sub-grafo completo correspondente ao clique maximal em que o o somatório do valor dos vértices seja máximo.

Neste trabalho é usada uma heurística para encontrar o melhor sub-conjunto de movimentos independentes, o Algoritmo 10 ilustra a heurística utilizada para fazer o merge dos movimentos da solução. O método apenas pode ser executado quando as soluções base são iguais para as duas soluções de entrada. Os movimentos são reunidos num conjunto M' e então ordenados conforme a melhoria que podem aplicar à solução. Em seguida

cada movimento é testado, em busca de conflitos, contra as demais movimentos, caso haja conflito este movimento é descartado, os movimentos foram dispostos de forma que os movimentos com pior resultado sobre a solução sejam testados primeiro, desta forma aqueles com melhores valores são deixados por último para que possam ser preservados caso possuam conflitos com algum outro.

Algoritmo 10 Combinando movimentos de soluções diferentes

```

1: função GDVND_MERGE(Solução:  $s_1$ , Solução:  $s_2$ )
2:   se SoluçãoBase( $s_1$ ) = SoluçãoBase( $s_2$ ) então  $\triangleright$  Somente soluções com a mesma
   solução base podem ser combinadas
3:    $M' \leftarrow \text{sorted}(\text{Movimentos}(s_1) \cup \text{Movimentos}(s_2))$   $\triangleright$  Movimentos ordenados
   pelo valor da melhoria na solução
4:   para  $i \leftarrow 0$  to  $\text{len}(M')$  faça
5:     para  $j \leftarrow (i + 1)$  to  $\text{len}(M')$  faça
6:       se  $m_i \nparallel m_j$  então
7:          $M' \leftarrow M' - m_i$ 
8:       break
9:   retorna SoluçãoBase( $s_1$ ),  $M'$ 
10: retorna  $\min(s_1, s_2)$ 

```

O retorno da iteração de cada vizinhança do GDVND passa a corresponder não a uma solução s' mas a uma tupla com a solução e os movimentos aplicados, assim temos a resposta como $(s', \{m_1^x, m_2^x, \dots, m_k^x\})$.

É importante ressaltar a diferença do Multi Improvement para o GDVND, o primeiro é uma estratégia de exploração de vizinhanças em que a busca local avança de uma solução para outra fazendo uso de uma composição de movimentos independentes $\{m_1^x, m_2^x, \dots, m_k^x\} \subset M^x$ pertencentes a uma vizinhança, no caso do GDVND os movimentos independentes são oriundos de todas as vizinhanças utilizadas pelo algoritmo. Ainda na analogia com o Multi Improvement, o GDVND poderia ser visto como um multi improvement em que a sua vizinhança é a união de todas as vizinhanças utilizadas no GDVND.

A mesma ideia pode ser aplicada para um conjunto de movimentos $M = \{m_1, m_2, m_3, \dots\}$, ditos independentes se para uma solução s qualquer e para todo subconjunto não-vazio $M' = \{m_1, m_2, m_3, \dots, m_k\} \subseteq M$ temos $\widehat{m}_1(s) + \widehat{m}_2(s) + \widehat{m}_3(s) + \dots + \widehat{m}_k(s) = \widehat{m}_1(m_2 \circ m_3 \circ \dots \circ m_k \circ s)$. Utilizaremos a letra I para denotar movimentos independentes, por exemplo, $I(\{m_3, m_5, m_9\})$.

3.3.1 Detecção movimentos independentes

Para detectar quais movimentos são independentes, uma estrutura de dados C para gerenciamento de conflitos foi proposta. No caso do problema em questão, a estrutura se trata de vetor com N bits, onde cada bit i indica se houve alguma alteração relativa ao cliente i .

Exemplo C : $|0|0|0|1|1|1|1|0|0|0|0|0|0|$

Assim, a estrutura C começa com todos bits zero, considerando uma solução de referência s . Cada movimento altera a estrutura C além de alterar a solução corrente, indicando quais posições estão comprometidas. Da mesma maneira, uma função d de

detecção de conflitos indica se um movimento é "aplicável" dada certa configuração da estrutura de conflitos.

Dessa forma, ao aplicar um movimento, se for necessário alterar um bit para 1 que já tenha valor 1 então o movimento atual é conflitante com o anterior ou algum dos anteriores.

Como podemos ver a seguir a detecção de conflitos para uma solução de tamanho $n = 13$, os movimentos $Swap_{1,3} \parallel Swap_{5,9}$, $Swap_{5,9} \parallel Swap_{3,7}$ mas $Swap_{1,3} \nparallel Swap_{3,7}$.

$C : |0|0|0|0|0|0|0|0|0|0|0|0|0| - Swap_{1,3}$

$C : |0|1|0|1|0|0|0|0|0|0|0|0|0| - Swap_{5,9}$

$C : |0|1|0|1|0|1|0|0|0|1|0|0|0| - Swap_{3,7}$

$C : |0|1|0|0|0|1|0|1|0|1|0|0|0| - \text{Conflito}$

A estrutura começa com todos os bits com valor zero, os bits em negrito indicam aqueles que serão alterados pelo movimento atual, na última linha o bit em vermelho indica a existência de um conflito ($Swap_{1,3} \nparallel Swap_{3,7}$).

3.3.2 Exemplo de execução

No DVND, várias buscas começam de uma mesma solução de referência e o melhor movimento (ou primeiro de melhora) é retornado como candidato à inclusão em um pool de movimentos, chamado Histórico. Este pool pode ser visto como um grafo não direcionado, no qual os vértices são movimentos e arestas indicam conflitos entre movimentos. Porém, setas especiais são utilizadas para marcar dependência entre movimentos. Se m_B depende de m_A , então m_B só pode ser aplicado se m_A for aplicado antes na solução. Escolhendo o maior (ou menor) conjunto independente neste grafo, respeitando os conflitos e dependências, resultará na melhor combinação de movimentos em um dado momento, considerando múltiplas estruturas de vizinhança simultaneamente.

A Figura 24 apresenta um exemplo do grafo de Histórico para uma execução do GDVND com três processos de busca.

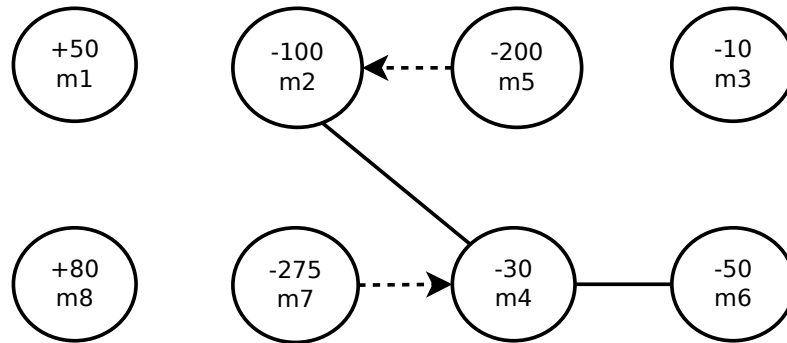


Figura 24 – Exemplo de um grafo de Histórico. A combinação ótima de custos consiste nos movimentos m_2 , m_3 , m_5 e m_6 .

Para entender melhor a ideia do algoritmo, a tabela abaixo simula a execução do algoritmo para três processos.

O Histórico começa distribuindo a solução s_1 para cada busca, e eventualmente coleta um novo movimento e distribui uma nova solução. Todos os cálculos são feitos sobre uma mesma solução de referência s_1 , que só é modificada no passo 20 do algoritmo. Nas iterações 7 e 9, note que existem conflitos a serem resolvidos (vide grafo na Figura 24). Após a iteração 19, o Histórico percebe que os três processos tem movimentos em comum:

Tabela 2 – Execução do GDVND

#	f	solução	direção	processo	conflito
1	1000	s_1	\rightarrow	P1	
2	1000	s_1	\rightarrow	P2	
3	1000	s_1	\rightarrow	P3	
4	970	$m_4 \circ s_1$	\leftarrow	P3	
5	970	$m_4 \circ s_1$	\rightarrow	P3	
6	900	$m_2 \circ s_1$	\leftarrow	P1	
7*	900	$m_2 \circ s_1$	\rightarrow	P1	$m_2 \nparallel m_4$
8	1050	$m_1 \circ s_1$	\leftarrow	P2	
9*	900	$m_2 \circ s_1$	\rightarrow	P2	
10	890	$m_3 \circ m_2 \circ s_1$	\leftarrow	P1	
11	890	$m_3 \circ m_2 \circ s_1$	\rightarrow	P1	$m_2 \parallel m_3$
12	700	$m_5 \circ m_2 \circ s_1$	\leftarrow	P2	
13	690	$m_5 \circ m_3 \circ m_2 \circ s_1$	\rightarrow	P2	$m_3 \parallel m_5$ e $m_2 \parallel m_5$
14	695	$m_7 \circ m_4 \circ s_1$	\leftarrow	P3	
15	685	$m_7 \circ m_4 \circ m_3 \circ s_1$	\rightarrow	P3	$m_2 \nparallel m_7$ e $m_5 \nparallel m_4$
16	840	$m_6 \circ m_3 \circ m_2 \circ s_1$	\leftarrow	P1	
17	640	$m_6 \circ m_3 \circ m_5 \circ m_2 \circ s_1$	\rightarrow	P1	$m_6 \parallel m_2, m_3, m_5$
18	765	$m_8 \circ m_7 \circ m_4 \circ m_3 \circ s_1$	\leftarrow	P3	
19	640	$m_6 \circ m_3 \circ m_5 \circ m_2 \circ s_1$	\rightarrow	P3	$m_8 \nparallel m_6$

m_3 , m_5 e m_2 . Então uma nova solução s_2 é armazenada no Histórico como solução de referência, sendo ela $s_2 = m_3 \circ m_5 \circ m_2 \circ s_1$ (de custo 690). Assim, os processos P1 e P3 continuam naturalmente processando a solução $m_6 \circ s_2$ (antiga $m_6 \circ m_3 \circ m_5 \circ m_2 \circ s_1$), e P2 continua com s_2 (antiga $m_5 \circ m_3 \circ m_2 \circ s_1$). Note também que, ao adotar estes três movimentos, todos movimentos com conflitos (ou dependência de algum movimento em conflito) tem de ser eliminados. Então, m_4 e m_7 são descartados. Vale observar que m_1 e m_8 nunca foram armazenados, pois eram de piora. O grafo final então consiste somente dos movimentos m_6 , m_3 , m_5 e m_2 .

Tabela 3 – Consolidação de uma nova solução no GDVND

#	f	solução	local
20	690	$s_2 = m_3 \circ m_5 \circ m_2 \circ s_1$	Histórico
	640	$m_6 \circ s_2$	P1
	690	s_2	P2
	640	$m_6 \circ s_2$	P3

3.3.3 Controle de execução entre CPU e GPU

Para entender como o método tira proveito da arquitetura híbrida CPU-GPU segue o presente exemplo: A CPU é utilizada para decidir qual solução cada processo de busca local irá utilizar, minimizando as transferências de memória CPU-GPU.

EXEMPLO:

Processo A \rightarrow CPU

Processo B \rightarrow Kernel GPU Swap

Processo C \rightarrow Kernel GPU 2-Opt

Processo D \rightarrow Kernel GPU Or-1

Processo E \rightarrow Kernel GPU Or-2

Processo F \rightarrow Kernel GPU Or-3

Solução $s_1 = [1,2,3,4,5,6,7,8,9,10]$, ($f(s_1) = 1000$).

A solução s_1 entra na GPU para todas as buscas B-F, cada uma em um fluxo (thread) distinto. Supomos que o processo B (Swap) termina antes com ganho -100 (melhora, $\widehat{m}_1 = -100$) na troca $m_1 = \text{Swap}_{2,5}$. A CPU então cria $s_2 = m_1 \circ s_1$. Como s_2 é a melhor solução atual, ela é copiada para a GPU, e o processo B roda em cima de $s_2 = [1,5,3,4,2,6,7,8,9,10]$, com $f(s_2) = 900$.

Neste momento, o processo D termina com $m_2 = \text{Or} - 1_{7,9}$, movendo cliente 7 para depois do 9, com ganho -50, $\widehat{m}_2 = -50$. A CPU então analisa m_1 e m_2 , como não há conflitos uma solução $s_3 = m_2 \circ m_1 s_1$ é criada para o processo D executar. $s_3 = [1,5,3,4,2,6,8,9,7,10]$, com $f(s_3) = 850$

O processo C termina com o movimento $m_3 = 2\text{Opt}_{1,3}$ (inverte de 1 a 3) com ganho -120 ($\widehat{m}_3 = -120$). A CPU percebe que $m_3 \nparallel m_1$, mas $m_3 \parallel m_2$. Então a solução $s_4 = m_4 \circ m_2 \circ s_1$ é gerada e passada à GPU para o processo C, com $f(s_4) = 830$

O processo E termina sem ganho com o movimento $m_4 = \text{Or} - 2_{1,4}$ (move clientes 1,2 para depois do 4). A CPU percebe que $m_4 \nparallel m_1$ e $m_4 \nparallel m_3$ mas $m_4 \parallel m_2$. Porém, a melhor configuração atual é s_4 ($f(s_4) = 830$), então a busca parte dela (e ela já está na memória da GPU).

Finalmente, o processo B (Swap) termina novamente com melhora -60 e troca $m_5 = \text{Swap}_{3,8}$. Esta troca roda em cima de s_2 , então não existem outros movimentos para haver conflito. A solução $s_5 = m_5 \circ s_2$ com $f(s_5) = 840$ é criada, porém, a solução s_4 ($f(s_4) = 830$) é melhor. A CPU então relança o processo B com a solução s_4 .

3.3.4 Passo iterativo

Utilizando o termo convencionado na seção 2.4.6, cada passo iterativo do GDVND retorna a melhor solução encontrada para todas as vizinhanças combinado com a melhor solução já encontrada. Dessa forma, sendo N a união de todas as vizinhanças, contudo a resposta do GDVND pode conter mais de um movimento combinados simultaneamente, assim convencionaremos chamar de $\mathcal{N} = \{m_1 \circ m_2 \circ \dots \circ m_z \circ s \mid \forall m_i \in \mathcal{M} \wedge \forall i, j \quad m_i \parallel m_j\}$, podemos ver, pela definição de \mathcal{N} , que $N \subset \mathcal{N}$ pois qualquer elemento de N pode ser escrito da forma $m_1 \circ s$ que também pertence a \mathcal{N} . Assim o passo iterativo do GDVND pode ser dado pela Equação 3.5.

$$\rho^{GDVND}(s) = s' \in \mathcal{N} \text{ sendo } f(s') < f(s''), \forall s'' \in \mathcal{N}(s) \text{ com } s'' \neq s \quad (3.5)$$

Fazendo uso da notação de movimentos podemos escrever 3.3 como a Equação 3.4.

$$\rho^{GDVND}(s) = m_1 \circ m_2 \circ \dots \circ m_z \circ s \text{ sendo } m \in \mathcal{M} \wedge \forall i, j \quad m_i \parallel m_j \text{ com } i \neq j \quad (3.6)$$

Pelo passo iterativo do RVND ρ^{RVND} (3.2), do DVND ρ^{DVND} (3.4) e do GDVND ρ^{GDVND} (3.6) podemos concluir que $\rho^{GDVND}(s) \leq \rho^{DVND}(s) \leq \rho^{RVND}(s)$, contudo novamente isso não é suficiente para afirmar que o GDVND encontre necessariamente melhores resultados pois o pode acabar convergindo muito cedo para um mínimo local.

À primeira vista pode parecer que pela vizinhança do passo iterativo do GDVND conter as vizinhanças do RVND e DVND contudo podemos ver que toda solução encontrada pelo GDVND pode ser gerada por um conjunto de operações do DVND ou RVND.

Para que uma solução produzida pelo GDVND não possa ser produzida pelo RVND ou DVND deve haver um movimento nessa solução que não seja possível encontrar com RVND ou DVND, suponhamos então que o GDVND produziu uma solução $m_1 \circ m_2 \circ \dots \circ m_z \circ s \in \mathcal{N}$ com $m_i \in \{m_1, m_2, \dots, m_z\}$ tal que $m_i \notin \mathcal{M}$.

Demonstração.

$$\begin{aligned} \rho^{GDVND}(s) &= m_1 \circ m_2 \circ \dots \circ m_i \circ \dots \circ m_z \circ s && \text{Por comutatividade} \\ \rho^{GDVND}(s) &= m_i \circ m_1 \circ m_2 \circ \dots \circ m_z \circ s && \text{Fazendo } s' = m_1 \circ m_2 \circ \dots \circ m_z \circ s \\ \rho^{GDVND}(s) &= m_i \circ s' && \text{Logo uma contradição } \perp \end{aligned}$$

□

Quando se escreve $\rho^{GDVND}(s) = m_i \circ s'$ chega-se a uma contradição pois, pela definição do passo iterativo do GDVND, temos que o movimento m_i deveria pertencer ao conjunto \mathcal{M} contudo supôs-se por hipótese que $m_i \notin \mathcal{M}$.

3.4 Vizinhanças

RVND, DVND e GDVND usam um conjunto de vizinhanças para processar a solução, foram escolhidas 5 estruturas diferentes, a saber, Swap, 2-Opt, OrOpt-1, OrOpt-2 e OrOpt-3, as mesmas utilizadas em (RIOS et al., 2016). No caso do DVND, a cada nova solução encontrada esta pode ser submetida para todas as vizinhanças simultaneamente (tentando explorar melhorias de vizinhanças diferentes simultaneamente). A intenção principal é prover uma implementação dataflow de um procedimento de busca local de muitas vizinhanças, o que é uma contribuição original para ambas comunidades, de otimização e computação paralela.

Para a busca local foi usado uma abordagem Multi Improvement (MI) (RIOS et al., 2016) em todas as vizinhanças, na qual é calculado os custos de todos os movimentos e os resultados armazenados num vetor. Usando a estratégia Best Improvement (BI) apenas a melhor solução é selecionada, na estratégia Multi Improvement todos os *movimentos independentes* são simultaneamente aplicados para a solução dada, provendo assim uma convergência mais rápida para o ótimo local.

A enumeração de todos as estratégias de vizinhança roda numa implementação em GPU, assim temos um algoritmo dataflow onde a implementação de cada nó *oper* roda em GPU para ambos os casos, a configuração do lançamento do kernel para cada estratégia de vizinhança é descrita na Tabela 4, estas configurações foram obtidas fazendo-se uso das bibliotecas fornecidas pela NVIDIA para otimização do uso dos recursos conforme o kernel que se deseja executar, conforme descrito em (NVIDIA, 2014) Mais detalhes sobre a implementação GPU da enumeração das vizinhanças podem ser encontrados em (RIOS et al., 2016).

3.4.1 Tabela de conflitos

Para classificação de movimentos independentes é necessário identificar quando um movimento aplicado a uma solução não conflita com outro movimento aplicado a mesma solução, conforme a definição estabelecida na Seção 2.4.2.3.

A identificação de conflitos pode ser feita em $\mathcal{O}(n)$ conforme descrito na Seção 3.3.1, contudo nos termos deste trabalho que trata do Problema da Mínima Latência (conforme descrito na Seção 2.3), para as vizinhanças *swap*, *2-opt* e *oropt-k* utilizadas, dois movimentos são independentes quando são satisfeitas as condições expressas na Tabela 5, desta forma é possível identificar a independência de movimentos em $\mathcal{O}(1)$ independente da solução base considerada.

Tabela 4 – Configuração de lançamento para os kernels das vizinhanças. Onde *Grid* refere-se a quantidade de grides usada, *Block* o tamanho de cada bloco de threads e *Shared* indica a quantidade de memória compartilhada utilizada pelas threads em cada bloco.

Instância	n	Vizinhança	Grid	Block	Shared
#0 berlin52	52	Swap	26	53	1060
		2-Opt	27	53	1060
		OrOpt-1	50	53	1060
		OrOpt-2	49	53	1060
		OrOpt-3	48	53	1060
#1 kroD100	100	Swap	50	101	2020
		2-Opt	51	101	2020
		OrOpt-1	98	101	2020
		OrOpt-2	97	101	2020
		OrOpt-3	96	101	2020
#2 pr226	226	Swap	113	224	4540
		2-Opt	114	224	4540
		OrOpt-1	224	224	4540
		OrOpt-2	223	224	4540
		OrOpt-3	222	224	4540
#3 lin318	318	Swap	159	256	6380
		2-Opt	160	256	6380
		OrOpt-1	316	256	6380
		OrOpt-2	315	256	6380
		OrOpt-3	314	256	6380
#4 TRP-S500-R1	501	Swap	250	502	10040
		2-Opt	251	502	10040
		OrOpt-1	499	502	10040
		OrOpt-2	498	502	10040
		OrOpt-3	497	502	10040
#5 d657	657	Swap	328	512	13160
		2-Opt	329	512	13160
		OrOpt-1	655	512	13160
		OrOpt-2	654	512	13160
		OrOpt-3	653	512	13160
#6 rat784	783	Swap	391	672	15680
		2-Opt	392	672	15680
		OrOpt-1	781	672	15680
		OrOpt-2	780	672	15680
		OrOpt-3	779	672	15680
#7 TRP-S1000-R1	1001	Swap	500	1002	20040
		2-Opt	501	1002	20040
		OrOpt-1	999	1002	20040
		OrOpt-2	998	1002	20040
		OrOpt-3	997	1002	20040

	$\text{swap}(\mathbf{i}_2, \mathbf{j}_2)$	$\mathbf{2} - \text{opt}(\mathbf{i}_2, \mathbf{j}_2)$	$\text{oropt} - \mathbf{k}_2(\mathbf{i}_2, \mathbf{j}_2)$
swap $(\mathbf{i}_1, \mathbf{j}_1)$	$(i_1 - i_2 > 1) \wedge$ $(i_1 - j_2 > 1) \wedge$ $(j_1 - i_2 > 1) \wedge$ $(j_1 - j_2 > 1)$	$[(i_1 < i_2 - 1) \vee$ $(i_1 > j_2 - 1)] \wedge$ $[(j_1 < i_2 - 1) \vee$ $(j_1 > j_2 + 1)]$	$(j_1 < \min(i_2, j_2) - 1) \vee$ $(i_1 > \max(i_2, j_2) + k_2) \vee$ $[(i_1 < \min(i_2, j_2) - 1) \wedge$ $(j_1 > \max(i_2, j_2) + k_2)]$
$\mathbf{2} - \text{opt}$ $(\mathbf{i}_1, \mathbf{j}_1)$	$[(i_2 < i_1 - 1) \vee$ $(i_2 > j_1 + 1)] \wedge$ $[(j_2 < i_1 - 1)] \vee$ $(j_2 > j_1 + 1)]$	$(j_1 < i_2 - 1) \vee$ $(i_1 > j_2 + 1) \vee$ $(j_2 > i_1 - 1) \vee$ $(i_2 > j_1 + 1)$	$(i_1 > \max(i_2, j_2) + k_2) \vee$ $(j_1 < \min(i_2, j_2) - 1)$
$\text{oropt} - \mathbf{k}_1$ $(\mathbf{i}_1, \mathbf{j}_1)$	$(j_2 < \min(i_1, i_2) - 1) \vee$ $(i_2 > \max(i_1, i_2) + k_1) \vee$ $[(i_2 < \min(i_1, i_2) - 1) \wedge$ $(j_2 > \max(i_1, i_2) + k_1)]$	$(j_2 < \min(i_1, j_1) - 1) \vee$ $(i_2 > \max(i_1, j_1) + k_1)$	$[\max(i_1, j_1) + k_1 < \min(i_2, j_2)] \vee$ $[\min(i_1, j_1) > \max(i_2, j_2) + k_2]$

Tabela 5 – Tabela de movimentos independentes (não conflitantes)

3.5 Decomposição de vizinhanças (Disaggregated Neighborhoods)

Pela Figura 18 podemos perceber que o paralelismo horizontal, isto é, aquele obtido pela utilização de mais máquinas, neste método está limitado a quantidade de vizinhanças utilizadas no método DVND. Esta limitação representa um problema pois para obter melhor utilização do paralelismo horizontal seria necessário também adicionar novas estruturas de vizinhanças, o que aumentaria o esforço computacional e não necessariamente traria ganho de performance ou na qualidade do valor da solução.

Contudo este problema pode ser facilmente solucionado pela decomposição das vizinhanças, isto é, no lugar de um nó *oper* representar toda uma estrutura de vizinhança, este passa a processar uma sub-parte desta.

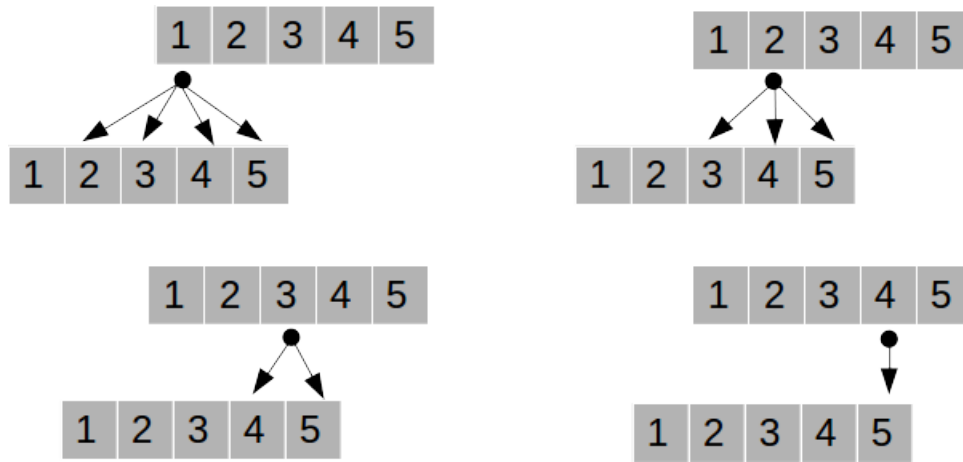


Figura 25 – Vizinhança com suas trocas para $n = 5$.

Isto posto, podemos separar a vizinhança em sub-grupos, um exemplo poderia ser a Figura 26 que mostra a mesma vizinhança da Figura 25 dividida em dois sub-grupos. Desta forma cada sub-grupo representaria um nó *oper* do diagrama dataflow DVND aumentando assim a escalabilidade horizontal do método.

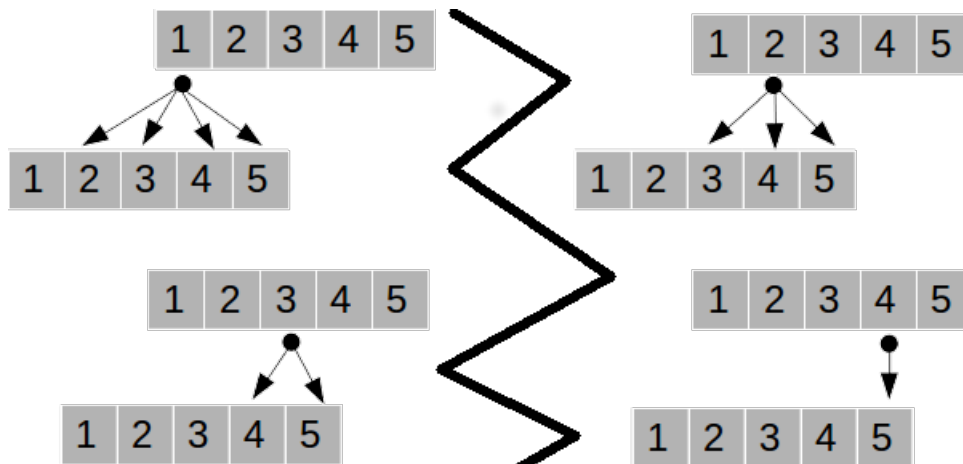


Figura 26 – Vizinhança com suas trocas dividida para $n = 5$.

A vizinhança na Figura 26 exemplifica a sub-divisão em dois grupos de vizinhança contudo esta divisão limita-se apenas ao tamanho da solução proporcionando um grande potencial para o desenvolvimento do paralelismo horizontal de qualquer vizinhança sem aumentar demasiadamente o custo computacional do método.

4 RESULTADOS

Este capítulo exhibe os resultados computacionais dos algoritmos propostos no Capítulo 3 para o caso do PML, para cada instância foi gerado um conjunto com 100 soluções iniciais aleatórias que foram submetidas aos métodos para comparação dos resultados.

Quando há referência à melhoria na solução (*Imp*), esta melhoria pode ser calculada pelo quociente do valor da solução inicial pela solução final, ou seja:

$$Imp = \frac{f(\text{solução inicial})}{f(\text{solução final})} \quad (4.1)$$

Desta forma quanto maior for o valor da melhoria (*Imp*) mais o método melhorou o valor da solução inicial.

4.1 Instâncias

Todas as instâncias usadas nos testes computacionais e cujas configurações de lançamento foram descritas na Tabela 4 são as mesmas usadas em (RIOS et al., 2016). Para o RVND foi feita uma implementação do algoritmo clássico (Algoritmo 5) e também a implementação dataflow mencionada na Figura 16 fazendo uso de uma máquina. Para o caso do DVND foi utilizada a implementação clássica (Algoritmo 6) e a implementação dataflow proposta (Figura 18), os resultados foram obtidos com diferentes números de máquinas e os mesmos são indicados conforme o caso.

4.2 Implementação e ambiente computacional

A implementação para cada algoritmo proposto no Capítulo 3 utiliza a linguagem de programação *C++11* em conjunto com a API CUDATM, para a implementação dos grafos e do ambiente dataflow foi utilizada a biblioteca Sucuri (ALVES et al., 2014)¹ implementada em Python, para a integração entre o dataflow e o código CUDA foi utilizada a biblioteca SimplePyCuda (SimplePyCuda, 2017)². As implementações com múltiplas threads usaram a biblioteca OpenMP. O ambiente computacional utilizado em todos os testes neste trabalho consiste de 4 máquinas com a seguinte configuração:

- Processador Intel®Core™i7-4820K 3.7 GHz (4 núcleos);
- 8 GB de memória RAM;
- Sistema Operacional Ubuntu 14.10 (x64);
- NVIDIA GeForce GTX 780 com 2304 CUDA cores.

¹Disponível em <<https://github.com/tiagoaoa/Sucuri>>

²Disponível em <<https://github.com/igormcoelho/simple-pycuda>>

4.3 Múltiplas portas de saída

Para validar a utilização de múltiplas portas de saída em cada nó do grafo dataflow, conforme discutido na Seção 3.2.3, foi realizado um experimento utilizando o dataflow DVND com apenas uma porta de saída (SOG) e comparado com outro experimento com o mesmo algoritmo DVND mas agora implementado utilizando múltiplas portas de saída para os nós (MOG).

O resultado para os tempos no experimento pode ser visto na Tabela 6, ao aplicar o teste de Wilcoxon pode se perceber que exceto para a instância 2 (de tamanho 226) em todas as outras situações houve significância estatística para verificar a diferença entre as amostras.

#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	MOG	4	52	2,43	4,028	2,698	2,811	2,928	2,953	0,429	0,0271
	SOG	4		2,329	3,786	2,68	2,761	2,87	2,802	0,257	
1	MOG	4	100	1,841	4,156	2,831	2,963	3,808	3,207	0,527	0,003571
	SOG	4		1,858	3,982	2,781	2,876	3,018	2,911	0,305	
2	MOG	4	226	2,504	4,545	3,079	3,171	3,29	3,243	0,355	0,1708
	SOG	4		2,643	4,481	3,021	3,263	3,405	3,242	0,299	
3	MOG	4	318	2,674	4,592	3,515	3,622	3,756	3,648	0,324	5,161e-06
	SOG	4		3,175	4,414	3,646	3,83	4,015	3,817	0,269	
4	MOG	4	501	3,423	5,801	4,345	4,514	4,691	4,496	0,38	4,901e-08
	SOG	4		3,781	6,382	4,553	4,783	4,974	4,783	0,38	
5	MOG	4	657	4,17	6,604	5,246	5,464	5,682	5,49	0,441	6,173e-07
	SOG	4		4,566	7,11	5,559	5,775	6,007	5,778	0,421	
6	MOG	4	783	5,434	8,442	6,305	6,508	6,691	6,514	0,507	1,429e-16
	SOG	4		5,95	8,881	6,782	7,04	7,356	7,087	0,471	
7	MOG	4	1001	7,871	10,88	8,836	9,09	9,493	9,164	0,566	7,009e-10
	SOG	4		8,253	10,96	9,352	9,626	10,05	9,693	0,565	

Tabela 6 – Tempos comparativos do SOG vs MOG onde SOG indica a execução com uma porta de saída e MOG com múltiplas portas de saída. Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

Nas Figuras 27a e 27b podemos ver uma sutil diferença favorecendo a versão sem múltiplas portas de saída (SOG), contudo a situação se inverte na instância 3.

Na Figura 27d referente à instância 3 (de tamanho 318) se mostra o início da melhoria no tempo pelo uso de múltiplas saídas nos nós do grafo dataflow.

O uso de múltiplas portas de saída, uma específica para cada nó de destino se mostra eficiente nas demais instâncias, podendo ser visto nas Figuras 28a-28d. A melhoria no tempo inicia-se na instância 3 de tamanho 318 e permanece por todas as instâncias seguintes.

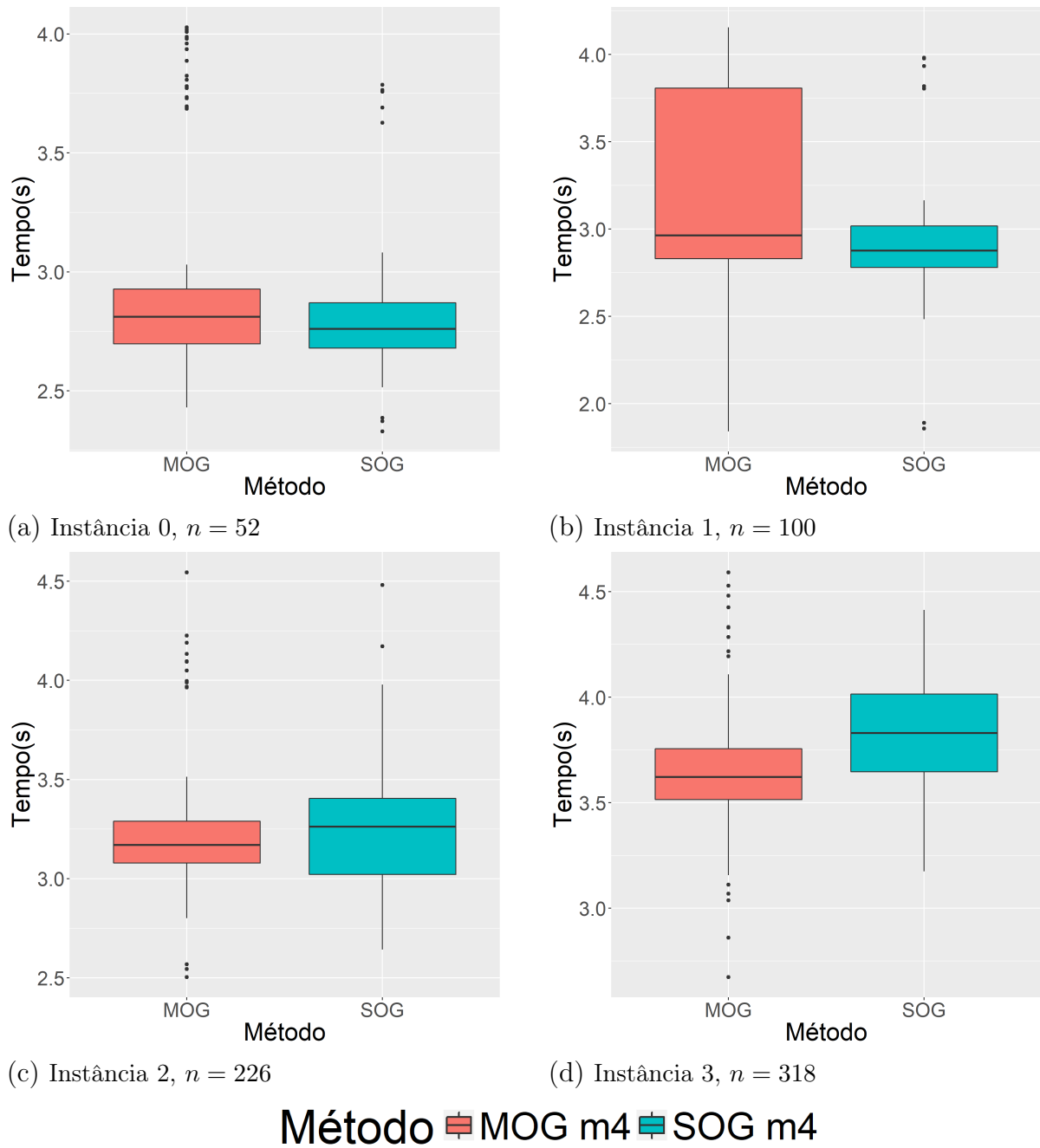


Figura 27 – Tempo do DVND, *SOG* refere-se a uma porta de saída e *MOG* a múltiplas portas de saída, n indica o tamanho, m indica o número de máquinas utilizadas. Instâncias 0 a 3.

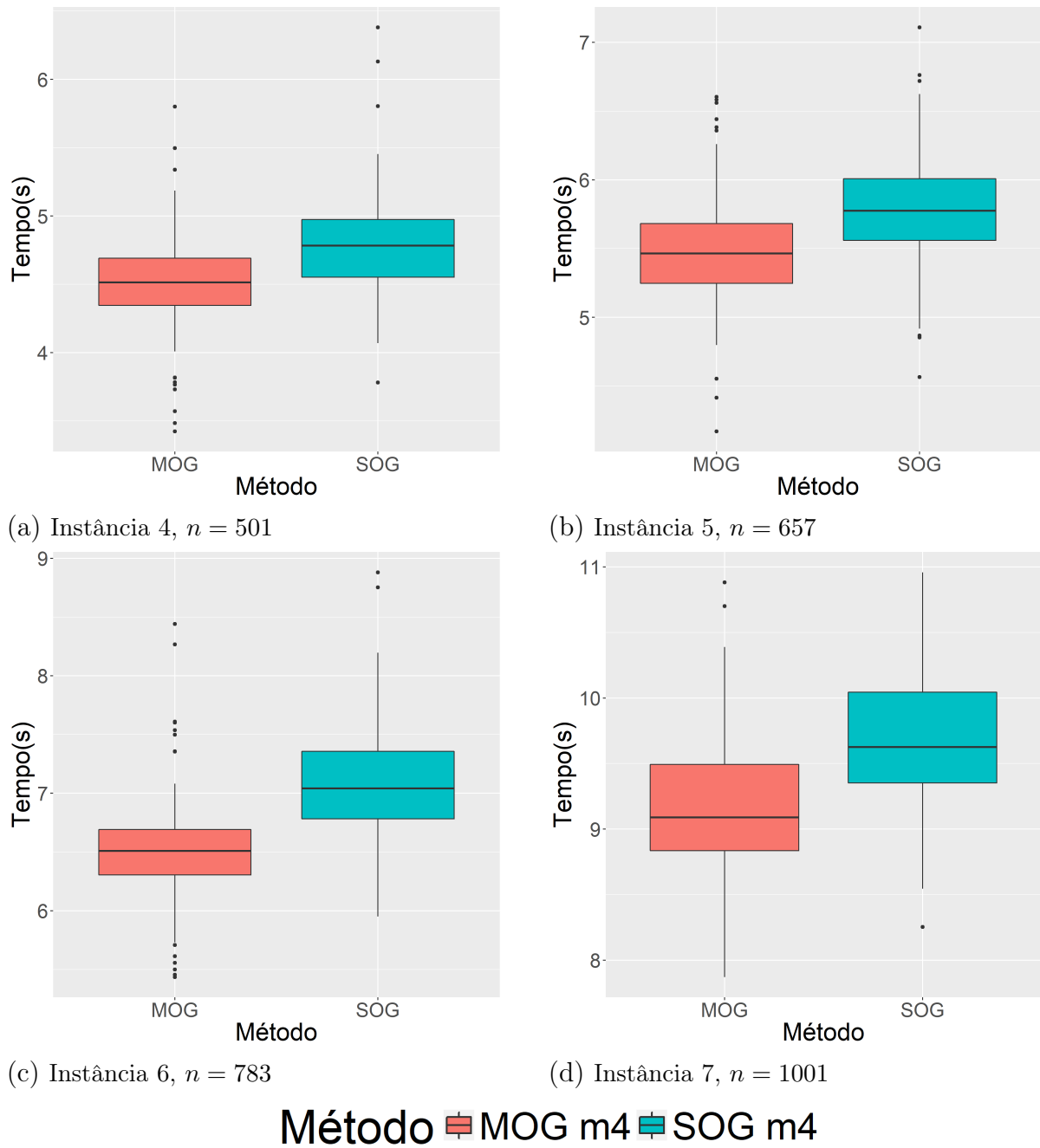


Figura 28 – Tempo do DVND, *SOG* refere-se a uma porta de saída e *MOG* a múltiplas portas de saída, n indica o tamanho, m indica o número de máquinas utilizadas. Instâncias 4 a 7.

4.4 RVND

Como veremos nas seções seguintes, apesar da implementação do RVND em dataflow não conseguir melhorar os tempos da implementação clássica do RVND o DVND clássico e DVND dataflow conseguem melhorar os tempos relativo ao tempo do RVND para as maiores instâncias.

4.4.1 Tempo

A Tabela 7 e as Figuras 29a-30d apresentam resultados para execuções utilizando apenas uma máquina ($m = 1$) pois pela construção naturalmente sequencial do RVND não utilizar paralelismo, logo o emprego de mais de uma máquina não traria ganhos em termos de desempenho tampouco no valor da solução.

#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DD	1	52	0,4269	1,77	1,258	1,416	1,531	1,321	0,323	9,279e-13
	DC	1		0,262	1,342	1,155	1,178	1,2	1,169	0,1	
1	DD	1	100	0,5236	2,773	0,8762	1,025	1,309	1,177	0,457	2,48e-07
	DC	1		0,3116	1,771	1,377	1,473	1,523	1,378	0,297	
2	DD	1	226	1,483	9,029	2,194	2,556	3,308	3,103	1,49	0,02545
	DC	1		1,413	6,555	2,565	2,989	3,616	3,106	0,926	
3	DD	1	318	1,983	7,007	3,133	3,541	4,125	3,824	1,07	2,559e-17
	DC	1		1,931	3,949	2,445	2,67	3,038	2,75	0,439	
4	DD	1	501	3,614	13,5	5,374	6,104	7,05	6,597	1,98	4,07e-13
	DC	1		3,63	7,018	4,697	4,987	5,52	5,095	0,59	
5	DD	1	657	6,878	22,9	9,351	10,12	11,74	11,43	3,82	1,202e-18
	DC	1		6,325	11,35	7,855	8,356	8,821	8,345	0,752	
6	DD	1	783	9,997	35,13	13,26	14,9	17,5	16,91	5,93	2,107e-15
	DC	1		10,38	15,38	11,7	12,49	13,04	12,5	1,05	
7	DD	1	1001	15,51	66,77	21,48	24,74	29,11	27,5	9,6	0,1547
	DC	1		19,8	30,42	22,71	24,31	25,73	24,27	2,13	

Tabela 7 – Tempos comparativos do RVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

Pela Tabela 7 e as Figuras 29a, 29d-30c podemos ver que o RVND em sua implementação clássica (RC) apresentou melhores tempos que o RVND na versão dataflow (RD).

Apenas no caso da instância 7, de tamanho 1001, representada pela Figura 30d, não houve diferença significativa, segundo o teste de Wilcoxon, para afirmar a existência de diferença nos dados.

4.4.2 Melhoria no valor da solução

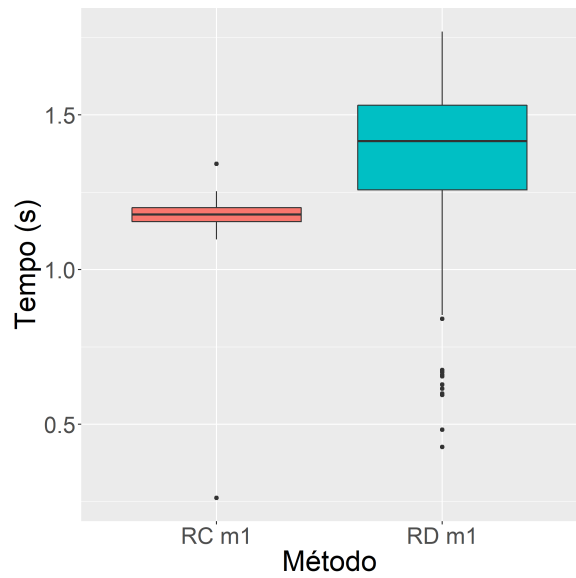
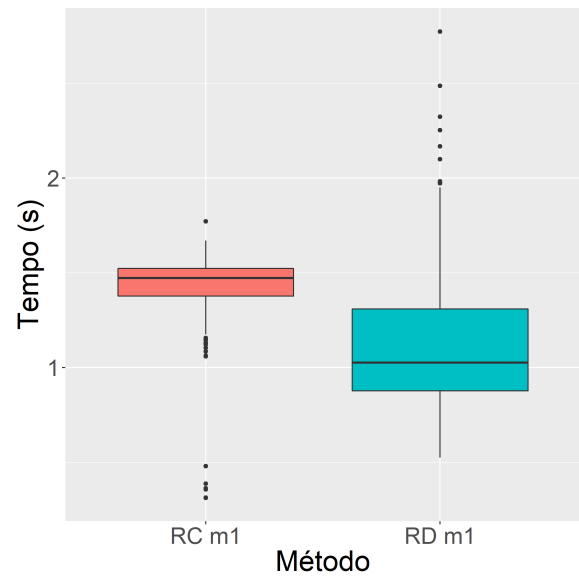
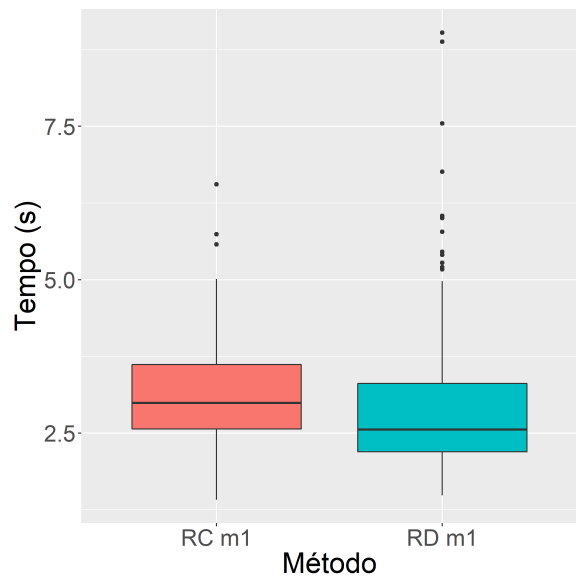
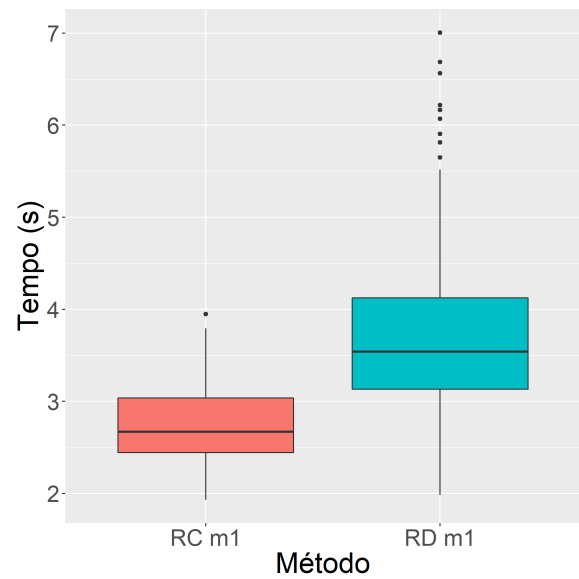
Em termos de melhoria no valor da solução (dada pela Equação 4.1), podemos ver Tabela 8 que não há grandes diferenças em termos de média (\bar{x}) nem de mediana (2Q) o

que se comprova nos resultados do teste de Wilcoxon que não apresenta diferença significativa senão nos resultados da instância 4 de tamanho 501.

#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DD	1	52	3,595	6,305	4,693	5,126	5,473	5,048	0,547	0,05327
	DC	1		3,501	6,265	4,82	5,332	5,614	5,173	0,624	
1	DD	1	100	6,163	9,343	7,686	8,125	8,317	8,049	0,58	0,5867
	DC	1		6,535	9,369	7,751	8,136	8,46	8,093	0,53	
2	DD	1	226	21,46	31,22	25,24	26,51	27,78	26,46	1,93	0,3557
	DC	1		22,88	31,07	25,64	26,55	27,88	26,77	1,73	
3	DD	1	318	13,8	17,03	14,79	15,24	15,61	15,21	0,587	0,8594
	DC	1		13,62	16,04	14,85	15,2	15,55	15,16	0,515	
4	DD	1	501	15,58	17,42	16,2	16,49	16,74	16,47	0,411	0,02205
	DC	1		15,11	17,17	16,07	16,37	16,61	16,33	0,412	
5	DD	1	657	18,1	20,84	19,07	19,4	19,79	19,42	0,52	0,4027
	DC	1		18,24	20,86	19,08	19,38	19,63	19,36	0,489	
6	DD	1	783	19,53	21,7	20,25	20,52	20,92	20,56	0,457	0,2132
	DC	1		19,37	21,83	20,16	20,42	20,75	20,48	0,485	
7	DD	1	1001	22,23	24,88	23,08	23,37	23,7	23,38	0,503	0,07059
	DC	1		22,42	24,52	23,02	23,22	23,46	23,27	0,401	

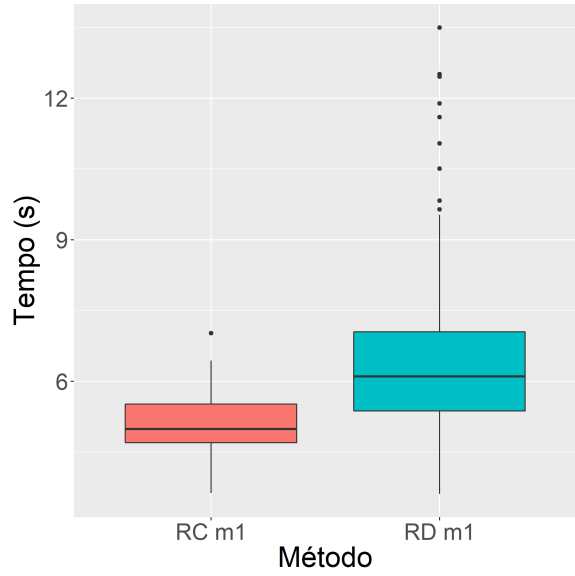
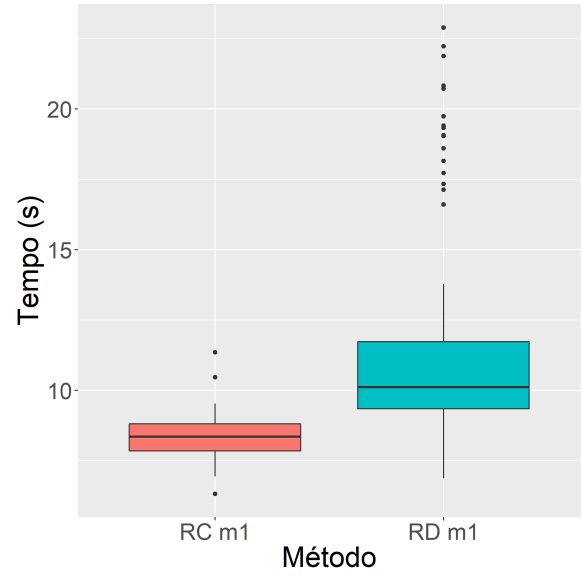
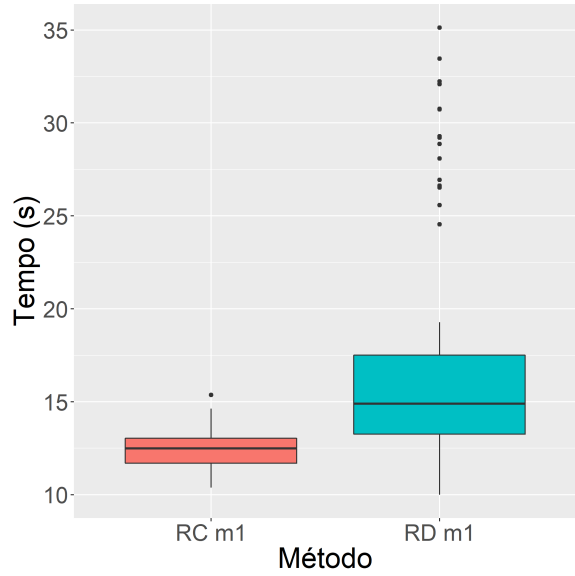
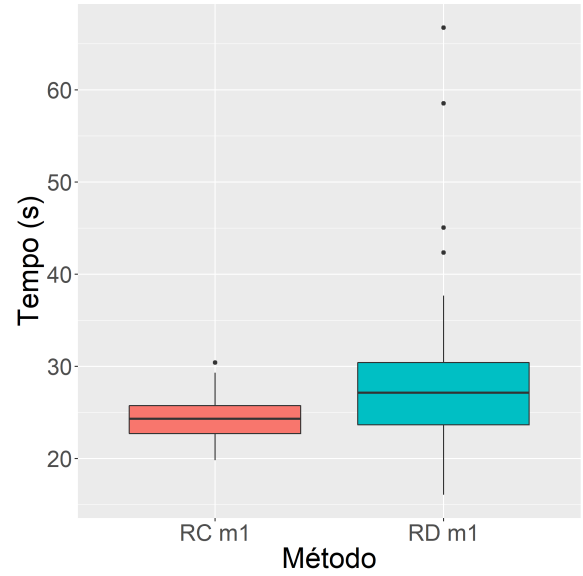
Tabela 8 – Comparativos de melhoria na solução para o RVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

Pelas Figuras 31a-32d se reforça a imagem de que o RVND clássico e o RVND implementado em dataflow apresentam resultados muito parecidos em termos de valor da solução encontrada. Este comportamento é esperado visto que, salvo pela aleatoriedade inerente à implementação sugerida por (SOUZA et al., 2010), ambas implementações cumprem a mesma tarefa.

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

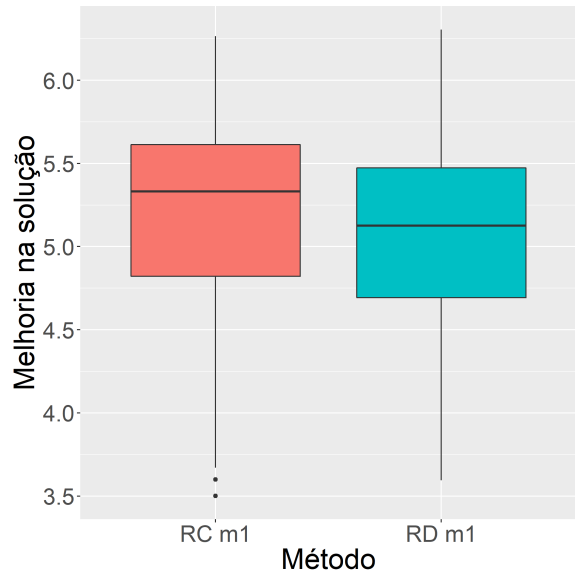
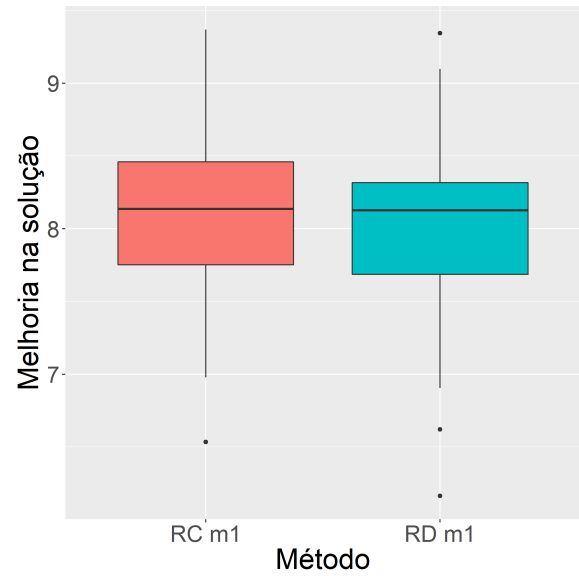
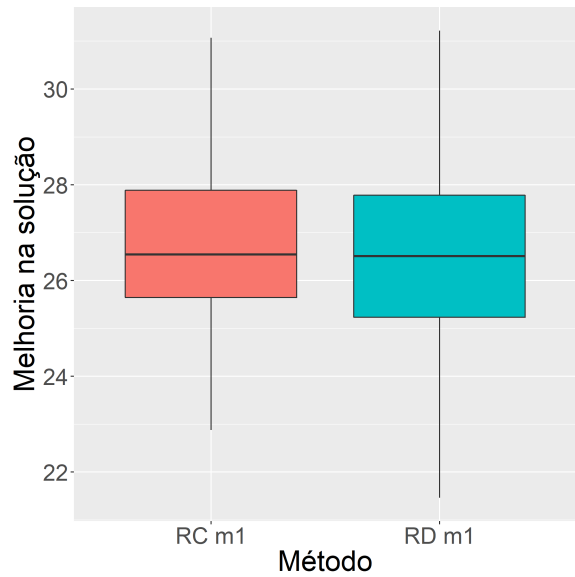
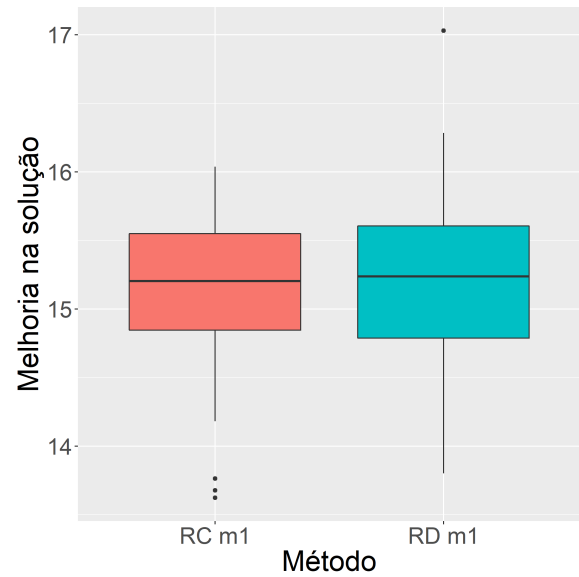
Método ■ RC m1 ■ RD m1

Figura 29 – Tempos do RVND, n representa o tamanho da instância, m indica o número de máquinas, RC refere-se ao RVND clássico e RD ao RVND implementado em dataflow. Instâncias 0 a 3.

(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

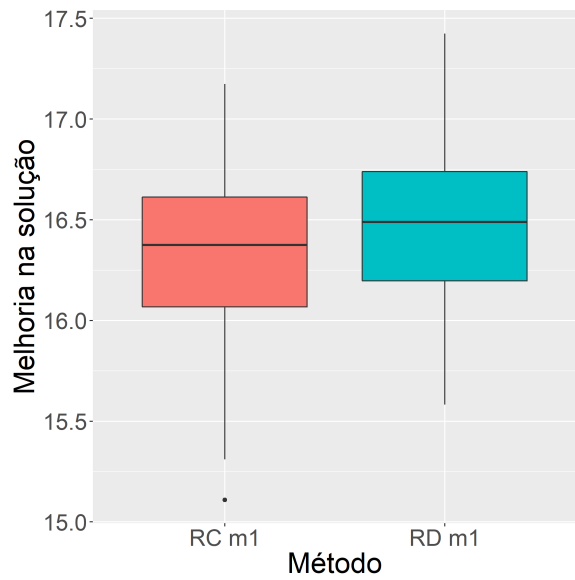
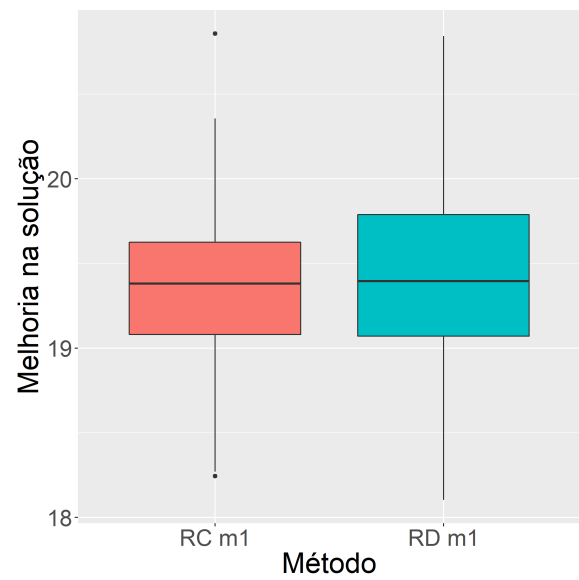
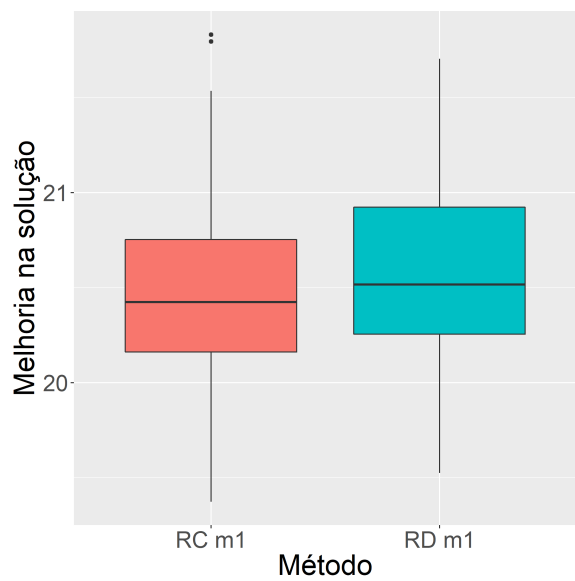
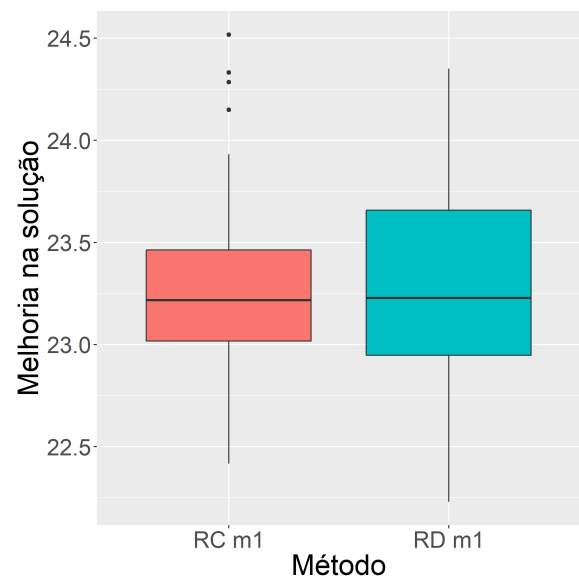
Método ■ RC m1 ■ RD m1

Figura 30 – Tempos do RVND, n representa o tamanho da instância, m indica o número de máquinas, RC refere-se ao RVND clássico e RD ao RVND implementado em dataflow. Instâncias 4 a 7.

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

Método ■ RC m1 ■ RD m1

Figura 31 – Melhoria no valor da solução para o RVND, n representa o tamanho da instância, m indica o número de máquinas, DC refere-se ao RVND clássico e DD ao RVND implementado em dataflow. Instâncias 0 a 3.

(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

Método ■ RC m1 ■ RD m1

Figura 32 – Melhoria no valor da solução para o RVND, n representa o tamanho da instância, m indica o número de máquinas, DC refere-se ao RVND clássico e DD ao RVND implementado em dataflow. Instâncias 4 a 7.

4.5 DVND

Para avaliar os resultados do DVND foi comparada a sua implementação clássica (DC) apresentada na literatura, apresentada no Algoritmo 6, com a implementação em dataflow (DD) apresentada na Figura 18. Os tempos de execução e melhoria na solução inicial são apresentados respectivamente na Tabela 9 e Tabela 10, as colunas destas designam o número da instância (#), tipo de implementação (Imp DC/DD), número de máquinas usado (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e o p-valor para o teste de Wilcoxon entre o a versão dataflow e a clássica.

4.5.1 Tempo

Pode-se ver pela Tabela 9 que o DVND clássico apresenta melhores tempos para as menores instâncias. Até a instância 4, de tamanho 501, os tempos do DVND clássico são sensivelmente melhores que os tempos do DVND em dataflow, contudo a partir da instância 5, de tamanho 657, a implementação em dataflow alcança os tempos da implementação clássica quando usa mais de uma máquina.

Na maior instância, de tamanho 1001, pode ser visto que o resultado da implementação dataflow para uma máquina é sutilmente melhor que da implementação clássica, o que se torna mais evidente ao utilizar mais de uma máquina, quando os tempos melhoram sensivelmente em relação à implementação clássica. Conforme se vê na Tabela 9 pela coluna p – *valor* há significância estatística para se verificar a diferença entre as amostragens.

Podemos ver na Figura 33a que o DVND clássico possui tempos bem menores que o DVND em dataflow e o uso de mais máquinas não consegue melhorar os tempos do procedimento.

A Figura 33b é bem parecida com a anterior, inclusive com tempos bastante próximos, indicando que o aumento de 52 para 100 no tamanho da solução não é suficiente para causar um grande aumento no tempos de solução pelo método.

Para a Figura 33c percebe-se que os tempos aumentam um pouco mas o comportamento é bastante semelhante, o DVND clássico é mais rápido para resolver o problema e aumentar o número de máquinas não melhora razoavelmente o desempenho.

Para a Figura 34a, que representa a instância 4 de tamanho 501, percebe-se pela primeira vez uma melhoria razoável no tempo do DVND em dataflow pelo uso de mais de uma máquina, contudo ainda não sendo suficiente para melhorar os resultado do DVND clássico.

Na instância 5, de tamanho 657, ilustrada na Figura 34b, o tempo do DVND em dataflow, quando usa mais de uma máquina, melhora alcançando ao DVND clássico.

Na instância 6, de tamanho 783, ilustrada na Figura 34c, os resultados são bastante parecidos com a instância anterior.

Na instância 7, de tamanho 1001, ilustrada na Figura 34d, os tempos alcançados pelo DVND dataflow são menores que o DVND clássico para mais de uma máquina alcançando assim melhores tempos para a maior instância.

4.5.2 Melhoria no valor da solução

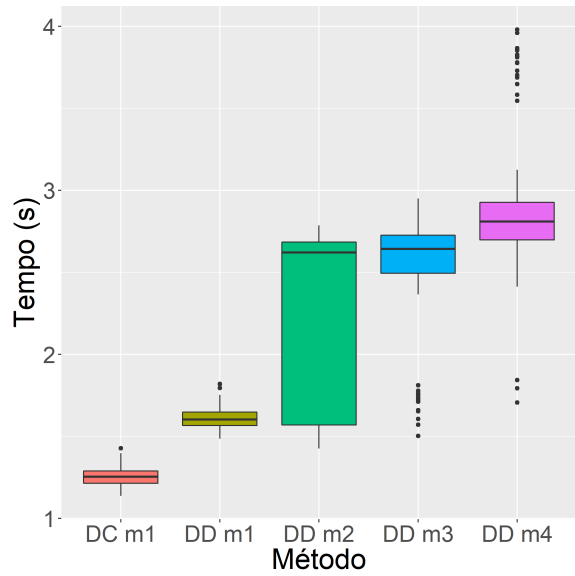
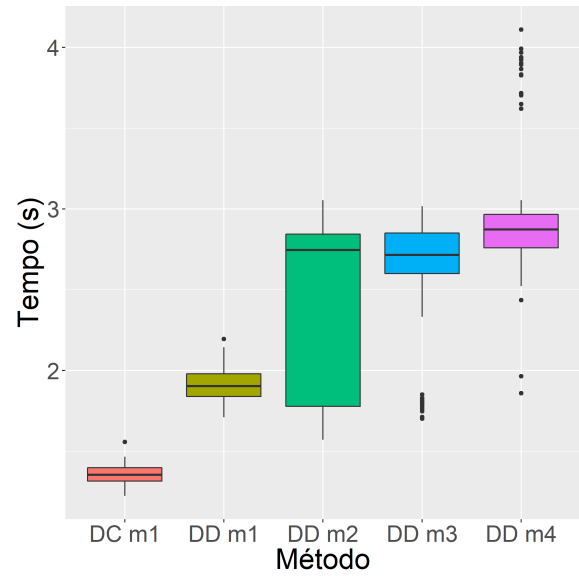
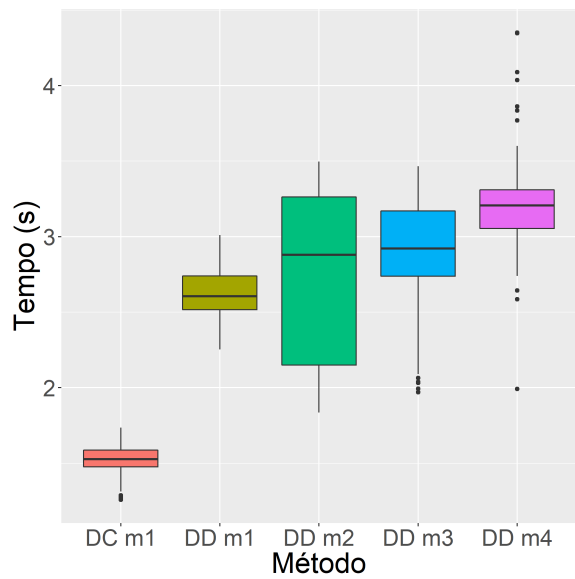
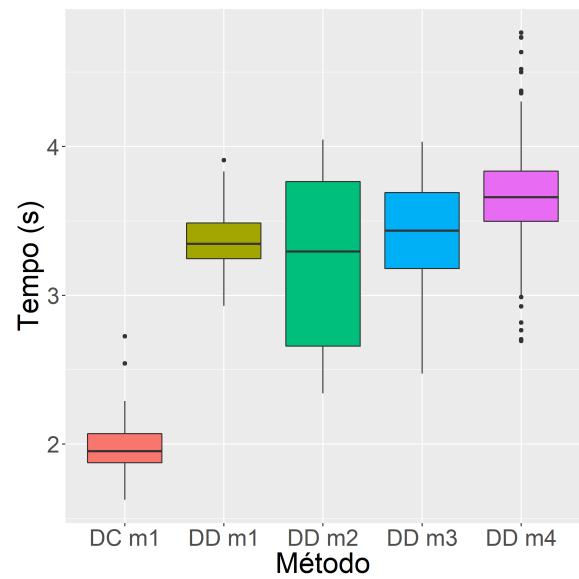
A Tabela 10 e as Figuras 35a-36d apresentam resultados em termos da melhoria no valor da solução inicial.

Como pode ser visto na Tabela 10, ficando mais evidente nas Figuras 35a-36d, em geral o DVND clássico (DC) consegue melhorar mais o valor da solução inicial quando comparado ao DVND dataflow (DD).

Ao aumentar o tamanho das instâncias o DVND clássico continua encontrando melhores resultados em termos de valor da solução mas também aumentando a variabilidade destes resultados, o que pode ser visto na Figura 35d, referente à instância 3 de tamanho 318, onde a amplitude interquartil do DVND clássico é de 2,88 número mais de 3 vezes o tamanho da maior amplitude interquartil para o DVND em dataflow para a mesma instância no valor de 0,86 para o DVND em dataflow com duas máquinas.

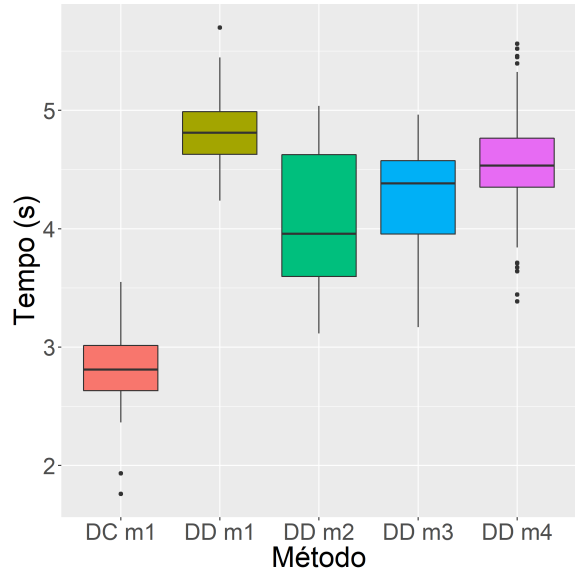
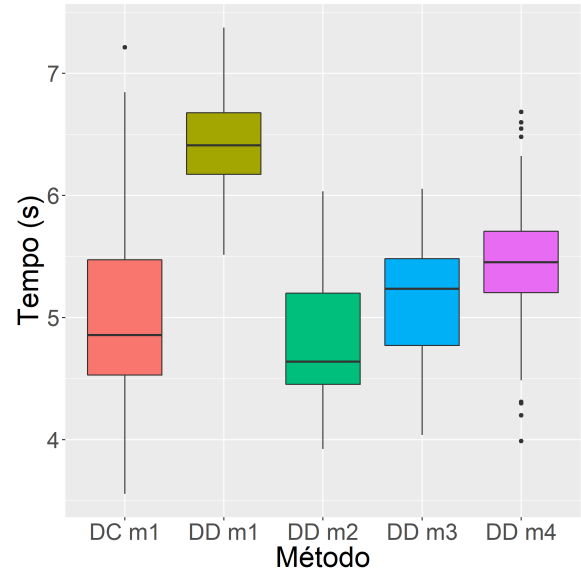
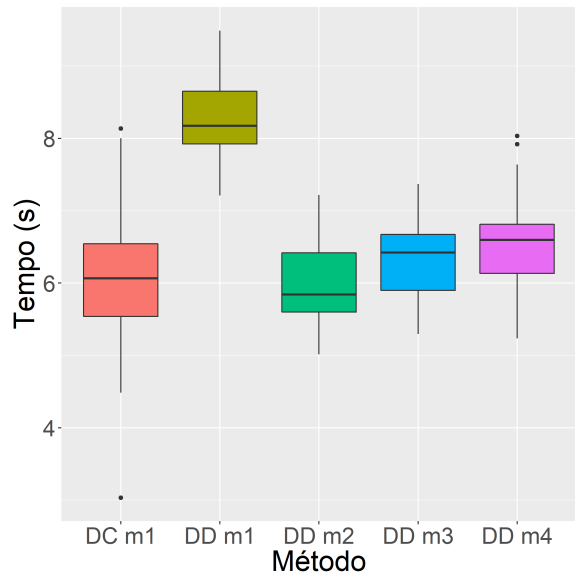
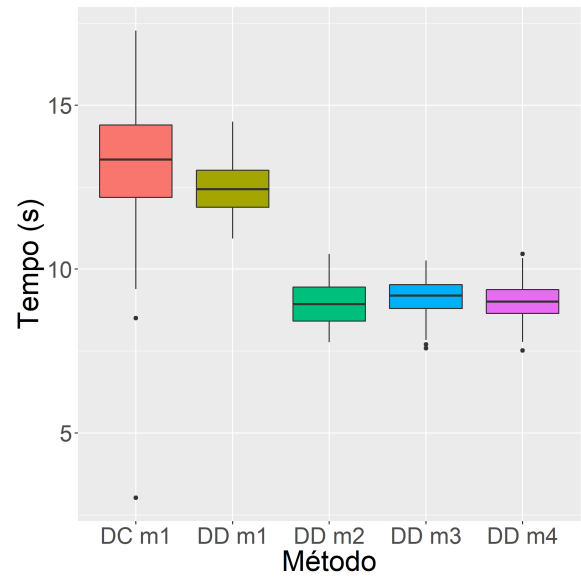
#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DD	1	52	1,487	1,82	1,566	1,603	1,648	1,61	0,067	1,238e-14
		2		1,425	2,787	1,569	2,621	2,684	2,218	0,551	1,238e-14
		3		1,504	2,95	2,495	2,643	2,727	2,497	0,386	1,238e-14
		4		1,706	3,98	2,698	2,81	2,926	2,91	0,44	1,238e-14
	DC	1		1,137	1,428	1,214	1,254	1,29	1,253	0,0554	
1	DD	1	100	1,711	2,195	1,839	1,903	1,981	1,918	0,104	1,238e-14
		2		1,572	3,055	1,778	2,745	2,844	2,422	0,532	1,238e-14
		3		1,702	3,018	2,601	2,716	2,85	2,606	0,369	1,238e-14
		4		1,86	4,111	2,76	2,873	2,966	2,968	0,417	1,238e-14
	DC	1		1,224	1,557	1,315	1,354	1,398	1,358	0,0583	
2	DD	1	226	2,253	3,011	2,517	2,605	2,74	2,621	0,155	1,238e-14
		2		1,835	3,497	2,149	2,879	3,262	2,74	0,543	1,238e-14
		3		1,969	3,465	2,738	2,921	3,17	2,881	0,389	1,238e-14
		4		1,991	4,351	3,053	3,206	3,31	3,212	0,315	1,238e-14
	DC	1		1,258	1,735	1,476	1,526	1,587	1,526	0,0937	
3	DD	1	318	2,929	3,909	3,247	3,345	3,487	3,352	0,2	1,238e-14
		2		2,341	4,047	2,659	3,295	3,764	3,233	0,542	1,238e-14
		3		2,474	4,032	3,181	3,434	3,691	3,39	0,37	1,238e-14
		4		2,693	4,767	3,497	3,66	3,834	3,662	0,425	1,238e-14
	DC	1		1,625	2,725	1,874	1,952	2,07	1,975	0,175	
4	DD	1	501	4,237	5,701	4,628	4,812	4,99	4,818	0,291	1,91e-14
		2		3,115	5,038	3,597	3,958	4,626	4,066	0,553	1,91e-14
		3		3,169	4,963	3,956	4,383	4,575	4,264	0,44	1,91e-14
		4		3,386	5,563	4,351	4,534	4,766	4,539	0,433	1,91e-14
	DC	1		1,76	3,549	2,632	2,811	3,014	2,842	0,315	
5	DD	1	657	5,512	7,375	6,173	6,41	6,677	6,425	0,369	0,03197
		2		3,923	6,032	4,451	4,639	5,2	4,822	0,525	0,03197
		3		4,038	6,053	4,77	5,235	5,482	5,12	0,482	0,03197
		4		3,988	6,683	5,204	5,453	5,706	5,421	0,498	0,03197
	DC	1		3,555	7,213	4,529	4,856	5,474	5,018	0,745	
6	DD	1	783	7,211	9,492	7,923	8,174	8,651	8,281	0,507	0,0001566
		2		5,016	7,217	5,598	5,841	6,42	5,99	0,523	0,0001566
		3		5,296	7,37	5,898	6,421	6,672	6,308	0,479	0,0001566
		4		5,237	8,034	6,134	6,598	6,815	6,496	0,588	0,0001566
	DC	1		3,031	8,138	5,537	6,066	6,543	6,105	0,848	
7	DD	1	1001	10,93	14,49	11,89	12,44	13,02	12,48	0,764	3,915e-11
		2		7,773	10,46	8,416	8,933	9,448	8,964	0,688	3,915e-11
		3		7,584	10,26	8,801	9,191	9,526	9,146	0,585	3,915e-11
		4		7,516	10,46	8,647	9,008	9,373	8,99	0,578	3,915e-11
	DC	1		3,024	17,28	12,19	13,34	14,39	13,28	1,92	

Tabela 9 – Tempos comparativos do DVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

Método ■ DC m1 ■ DD m1 ■ DD m2 ■ DD m3 ■ DD m4

Figura 33 – Tempo do DVND, n representa o tamanho da instância, m indica o número de máquinas, *DC* refere-se ao DVND clássico e *DD* ao DVND implementado em dataflow. Instâncias 0 a 3.

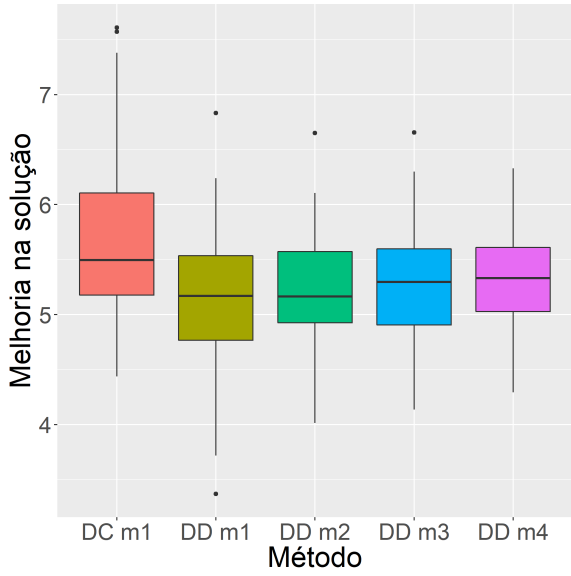
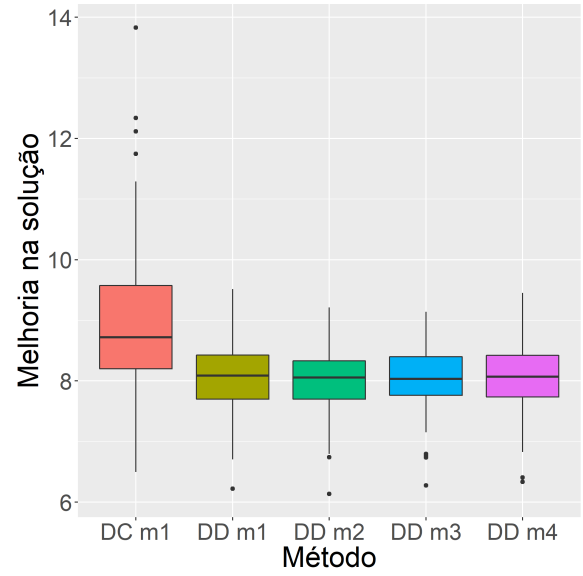
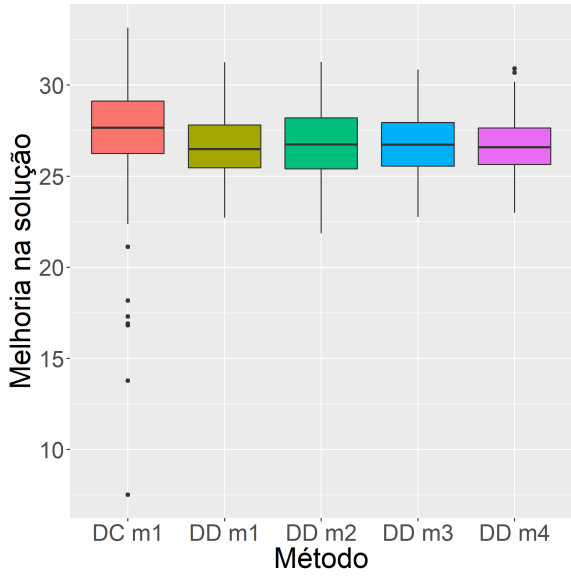
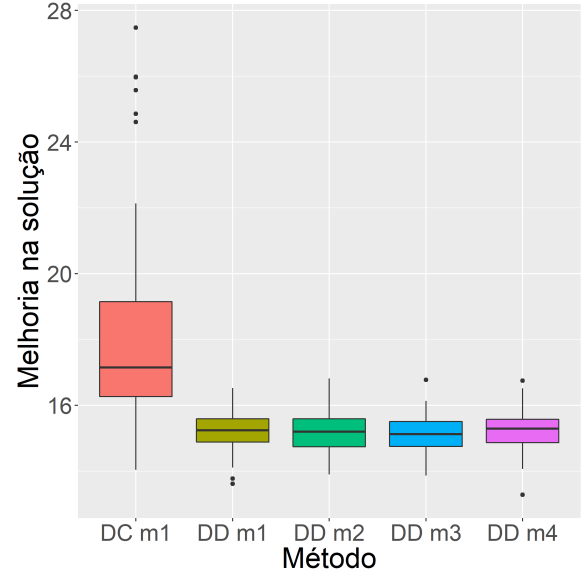
(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

Método ■ DC m1 ■ DD m1 ■ DD m2 ■ DD m3 ■ DD m4

Figura 34 – Tempo do DVND, n representa o tamanho da instância, m indica o número de máquinas, DC refere-se ao DVND clássico e DD ao DVND implementado em dataflow. Instâncias 4 a 7.

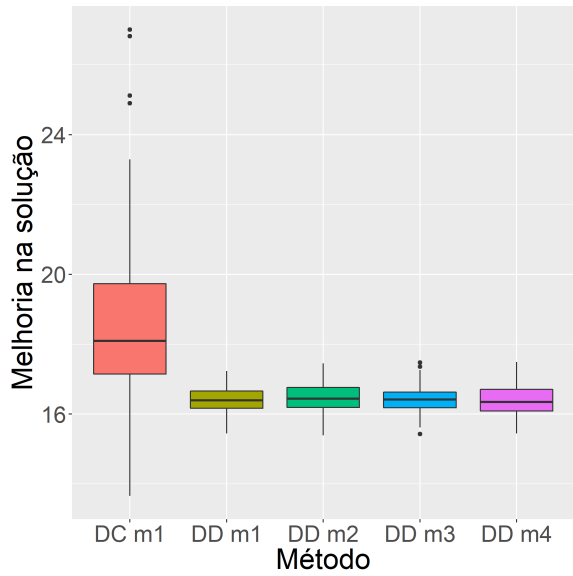
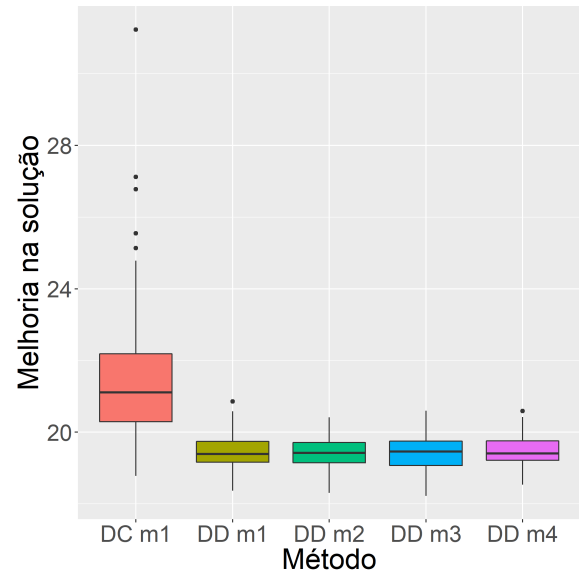
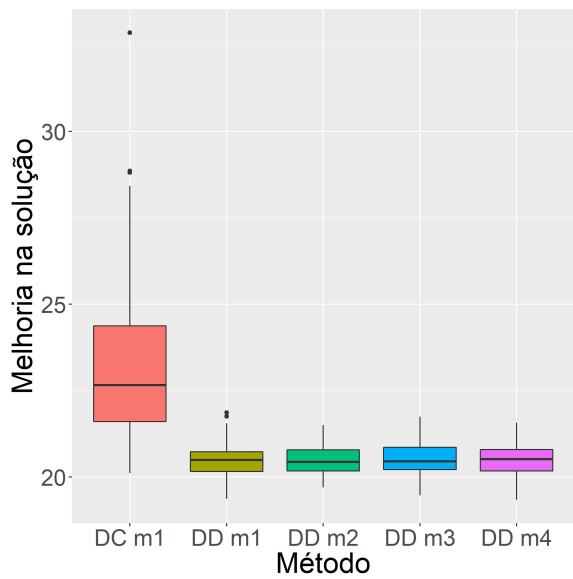
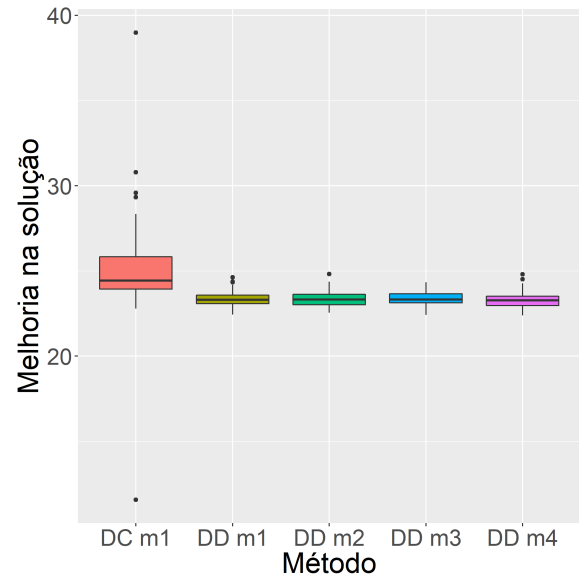
#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DD	1	52	3,368	6,832	4,765	5,17	5,535	5,104	0,622	0,007459
		2		4,013	6,65	4,926	5,163	5,572	5,201	0,513	0,007459
		3		4,135	6,655	4,906	5,296	5,597	5,283	0,505	0,007459
		4		4,293	6,329	5,026	5,332	5,611	5,316	0,455	0,007459
	DC	1		4,436	7,611	5,176	5,497	6,106	5,657	0,694	
1	DD	1	100	6,22	9,514	7,7	8,089	8,428	8,05	0,583	3,84e-06
		2		6,135	9,211	7,698	8,057	8,331	8,012	0,539	3,84e-06
		3		6,277	9,14	7,763	8,034	8,398	8,038	0,507	3,84e-06
		4		6,334	9,451	7,735	8,068	8,423	8,072	0,564	3,84e-06
	DC	1		6,496	13,83	8,201	8,72	9,575	9,005	1,18	
2	DD	1	226	22,73	31,25	25,46	26,48	27,81	26,72	1,77	0,0144
		2		21,87	31,28	25,4	26,74	28,2	26,78	2,03	0,0144
		3		22,76	30,84	25,55	26,72	27,94	26,75	1,71	0,0144
		4		22,99	30,9	25,64	26,6	27,65	26,65	1,66	0,0144
	DC	1		7,518	33,16	26,24	27,66	29,12	27,11	3,81	
3	DD	1	318	13,61	16,52	14,88	15,24	15,59	15,21	0,555	1,525e-13
		2		13,9	16,81	14,73	15,19	15,59	15,16	0,569	1,525e-13
		3		13,86	16,77	14,75	15,12	15,51	15,15	0,559	1,525e-13
		4		13,28	16,75	14,87	15,29	15,57	15,24	0,573	1,525e-13
	DC	1		14,04	27,48	16,27	17,15	19,15	17,91	2,65	
4	DD	1	501	15,44	17,23	16,16	16,39	16,66	16,4	0,374	7,427e-10
		2		15,39	17,45	16,19	16,44	16,76	16,44	0,404	7,427e-10
		3		15,43	17,47	16,18	16,41	16,63	16,43	0,429	7,427e-10
		4		15,44	17,49	16,09	16,35	16,71	16,39	0,419	7,427e-10
	DC	1		13,66	27,01	17,14	18,09	19,74	18,64	2,27	
5	DD	1	657	18,37	20,86	19,16	19,39	19,74	19,45	0,461	1,784e-10
		2		18,31	20,42	19,14	19,42	19,71	19,44	0,443	1,784e-10
		3		18,22	20,59	19,07	19,46	19,75	19,41	0,486	1,784e-10
		4		18,54	20,59	19,22	19,41	19,76	19,46	0,419	1,784e-10
	DC	1		18,78	31,23	20,29	21,11	22,19	21,49	1,9	
6	DD	1	783	19,37	21,86	20,16	20,49	20,74	20,48	0,486	9,465e-13
		2		19,7	21,51	20,18	20,44	20,79	20,5	0,439	9,465e-13
		3		19,47	21,75	20,22	20,46	20,86	20,51	0,473	9,465e-13
		4		19,34	21,58	20,18	20,52	20,8	20,5	0,453	9,465e-13
	DC	1		20,12	32,86	21,6	22,66	24,37	23,27	2,26	
7	DD	1	1001	22,44	24,62	23,08	23,3	23,57	23,34	0,41	5,073e-12
		2		22,54	24,81	23,01	23,32	23,62	23,33	0,441	5,073e-12
		3		22,42	24,34	23,14	23,32	23,65	23,37	0,38	5,073e-12
		4		22,39	24,8	22,97	23,27	23,52	23,27	0,44	5,073e-12
	DC	1		11,57	38,99	23,94	24,44	25,83	25,05	2,54	

Tabela 10 – Comparativos de melhoria na solução para o DVND na implementação clássica (DC) e a proposta de implementação usando dataflow (DD). Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

Método ■ DC m1 ■ DD m1 ■ DD m2 ■ DD m3 ■ DD m4

Figura 35 – Melhoria no valor da solução do DVND, n representa o tamanho da instância, m indica o número de máquinas, DC refere-se ao DVND clássico e DD ao DVND implementado em dataflow. Instâncias 0 a 3.

(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

Método ■ DC m1 ■ DD m1 ■ DD m2 ■ DD m3 ■ DD m4

Figura 36 – Melhoria no valor da solução do DVND, n representa o tamanho da instância, m indica o número de máquinas, DC refere-se ao DVND clássico e DD ao DVND implementado em dataflow. Instâncias 4 a 7.

4.6 GDVND

4.6.1 Tempo

Pode ser visto na Tabela 11 e nas Figuras 37a-38d que o DVND e GDVND apresentam comportamentos semelhantes em relação ao RVND, os tempos do RVND são melhores para as instâncias até o tamanho 318 e da instância 4 (tamanho 501) em diante o RVND começa a levar mais tempo para encontrar a resposta.

#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DVND	4	52	2,43	4,028	2,698	2,811	2,928	2,953	0,429	7,588e-05
	RVND	1		0,4269	1,77	1,258	1,416	1,531	1,321	0,323	3,564e-34
	GDVND	4		1,6	4,955	2,856	3,069	3,57	3,182	0,572	
1	DVND	4	100	1,841	4,156	2,831	2,963	3,808	3,207	0,527	2,201e-06
	RVND	1		0,5236	2,773	0,8762	1,025	1,309	1,177	0,457	3,673e-34
	GDVND	4		2,346	5,178	3,078	3,367	4,061	3,495	0,597	
2	DVND	4	226	2,504	4,545	3,079	3,171	3,29	3,243	0,355	7,31e-21
	RVND	1		1,483	9,029	2,194	2,556	3,308	3,103	1,49	2,196e-13
	GDVND	4		2,347	5,639	3,579	3,858	4,532	4,011	0,657	
3	DVND	4	318	2,674	4,592	3,515	3,622	3,756	3,648	0,324	1,74e-18
	RVND	1		1,983	7,007	3,133	3,541	4,125	3,824	1,07	8,397e-09
	GDVND	4		2,265	6,533	4,091	4,401	4,942	4,46	0,763	
4	DVND	4	501	3,423	5,801	4,345	4,514	4,691	4,496	0,38	2,401e-26
	RVND	1		3,614	13,5	5,374	6,104	7,05	6,597	1,98	0,8022
	GDVND	4		2,487	8,283	5,697	6,173	6,7	6,092	0,933	
5	DVND	4	657	4,17	6,604	5,246	5,464	5,682	5,49	0,441	6,681e-34
	RVND	1		6,878	22,9	9,351	10,12	11,74	11,43	3,82	1,967e-20
	GDVND	4		5,796	10,71	7,5	8,074	8,605	8,097	0,876	
6	DVND	4	783	5,434	8,442	6,305	6,508	6,691	6,514	0,507	4,018e-34
	RVND	1		9,997	35,13	13,26	14,9	17,5	16,91	5,93	1,396e-29
	GDVND	4		7,187	13,09	9,821	10,61	11,15	10,49	1,1	
7	DVND	4	1001	7,871	10,88	8,836	9,09	9,493	9,164	0,566	8,482e-34
	RVND	1		15,51	66,77	21,48	24,74	29,11	27,5	9,6	1,705e-30
	GDVND	4		9,195	19,53	15,71	16,47	17,22	16,35	1,54	

Tabela 11 – Tempos comparativos do GDVND com DVND e RVND. Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

4.6.2 Melhoria no valor da solução

A Tabela 12 mostra e as Figuras 39a-40d evidenciam que de forma geral não há grande diferença na qualidade da solução encontrada pelos métodos. De fato, ao analisarmos o p -valor encontrado pelo teste de Wilcoxon realizado da amostra do GDVND com as demais, do DVND e RVND podemos ver que para muitos casos não há significância estatística para afirmar a diferença entre as amostras, valores destacados em negrito na coluna $p - valor$ da Tabela 12.

#	Tipo	m	n	min	max	1Q	2Q	3Q	\bar{x}	σ	$p - valor$
0	DVND	4	52	4,27	6,401	4,954	5,287	5,564	5,264	0,475	2,447e-08
	RVND	1		3,595	6,305	4,693	5,126	5,473	5,048	0,547	0,0002838
	GDVND	4		2,706	6,457	4,024	4,836	5,273	4,609	0,871	
1	DVND	4	100	6,884	9,227	7,743	8,025	8,411	8,05	0,526	0,00738
	RVND	1		6,163	9,343	7,686	8,125	8,317	8,049	0,58	0,008289
	GDVND	4		4,012	9,348	7,256	7,838	8,311	7,627	1,04	
2	DVND	4	226	22,47	31,19	25,48	26,69	27,78	26,7	1,77	0,06525
	RVND	1		21,46	31,22	25,24	26,51	27,78	26,46	1,93	0,1826
	GDVND	4		5,265	30,89	24,97	26,17	27,62	25,29	4,31	
3	DVND	4	318	13,39	16,69	14,86	15,16	15,63	15,19	0,585	0,1916
	RVND	1		13,8	17,03	14,79	15,24	15,61	15,21	0,587	0,1484
	GDVND	4		5,163	16,94	14,67	15,07	15,5	14,38	2,68	
4	DVND	4	501	15,23	17,47	16,14	16,4	16,62	16,37	0,399	0,185
	RVND	1		15,58	17,42	16,2	16,49	16,74	16,47	0,411	0,005406
	GDVND	4		6,112	17,56	16,02	16,32	16,59	15,9	2,03	
5	DVND	4	657	18,22	20,49	19,08	19,42	19,71	19,38	0,476	0,3123
	RVND	1		18,1	20,84	19,07	19,4	19,79	19,42	0,52	0,1671
	GDVND	4		17,63	20,49	18,96	19,26	19,71	19,3	0,515	
6	DVND	4	783	19,44	21,83	20,16	20,48	20,85	20,52	0,469	0,1648
	RVND	1		19,53	21,7	20,25	20,52	20,92	20,56	0,457	0,05223
	GDVND	4		18,73	21,45	20,11	20,37	20,74	20,4	0,513	
7	DVND	4	1001	22,29	24,33	23,04	23,35	23,62	23,35	0,441	0,02061
	RVND	1		22,23	24,88	23,08	23,37	23,7	23,38	0,503	0,009164
	GDVND	4		20,66	24,09	22,91	23,17	23,5	23,17	0,493	

Tabela 12 – Comparativos de melhoria na solução para o GDVND com DVND e RVND. Instância (#), tipo de implementação (Tipo), número de máquinas (m), tamanho da instância (n), valor mínimo (min), máximo (max), primeiro quartil (1Q), mediana (2Q), terceiro quartil (3Q), média (\bar{x}), desvio padrão (σ) e p-valor para o teste de Wilcoxon entre as versões (valores em negrito quando $p - valor > 0.05$).

4.6.3 Analisando o tempo para combinar movimentos

Os tempos de execução do *GDVND* comparados com o *DVND* são exibidos nas Figuras 41a-42d, onde podemos ver o tempo gasto pelo *GDVND*, o tempo gasto pelo *DVND* e o tempo gasto pelo *GDVND* subtraído do tempo gasto para combinar os movimentos retornados pela busca local (*GDVND-MAN*), logo este último representa o tempo efetivamente gasto na busca local.

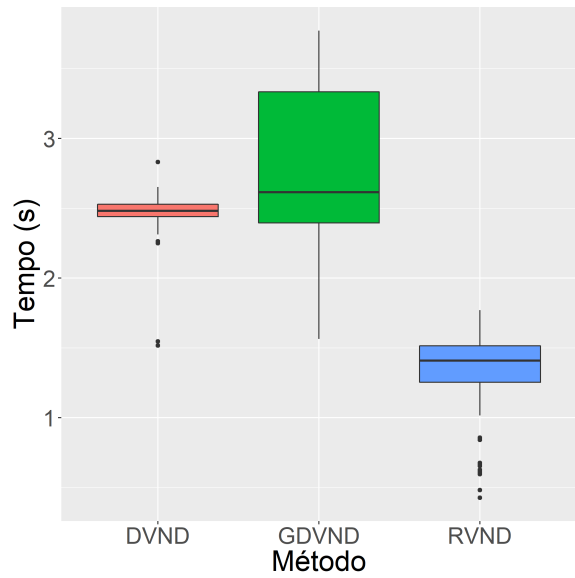
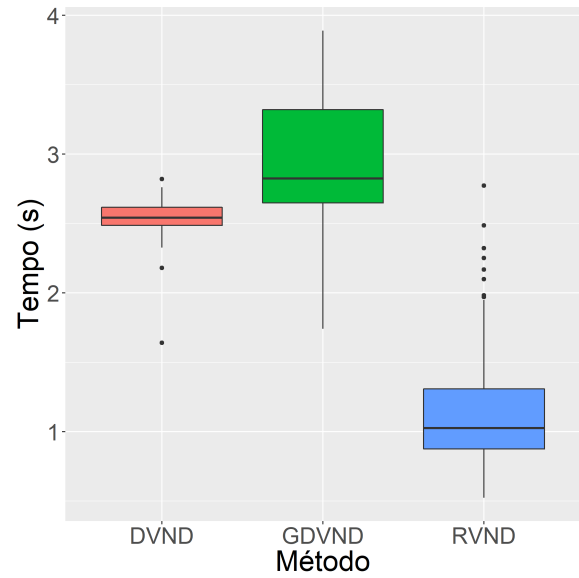
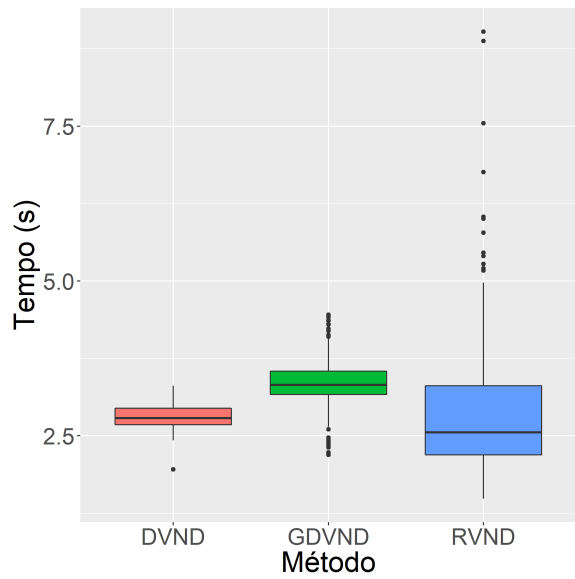
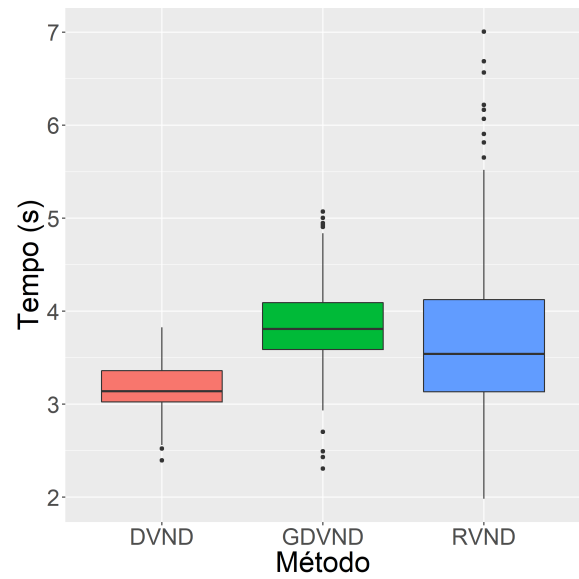
De forma geral podemos ver que, exceto para as duas primeiras instâncias (Figura 41a e 41b), o *GDVND* apresenta tempos de execução maiores que o *DVND* para a maioria das amostras.

Para as instâncias até o tamanho de 318 o tempo necessário para combinar os movimentos não representa grande diferença no tempo total de execução de forma que apenas a partir da instância 4 de tamanho 501 (Figura 42a) que o tempo sem as operações sobre os movimentos (*GDVND-MAN*) consegue ser melhor em uma quantidade maior de amostras.

No caso da instância 5 (Figura 42b) a diferença representada pelo tempo de execução

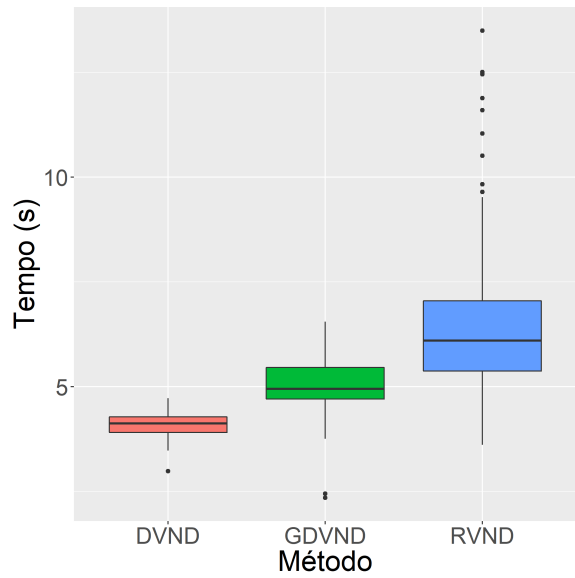
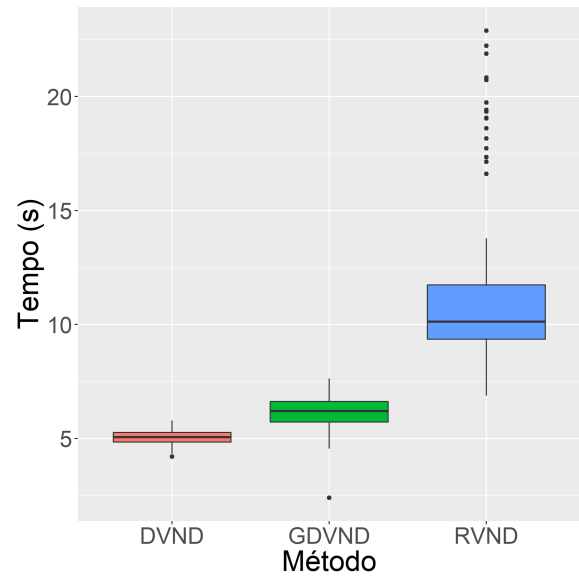
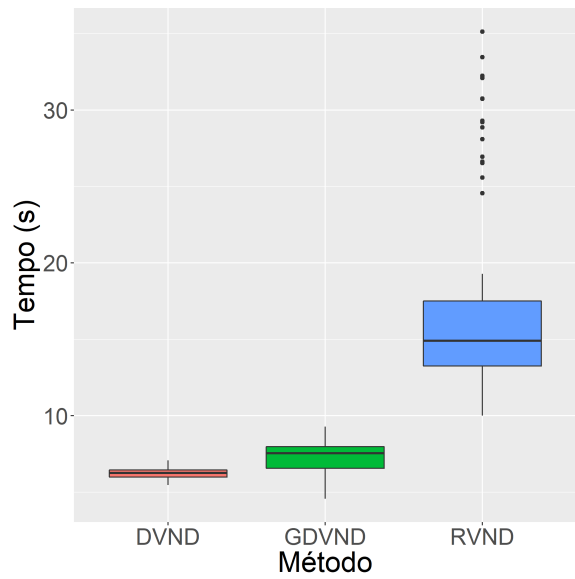
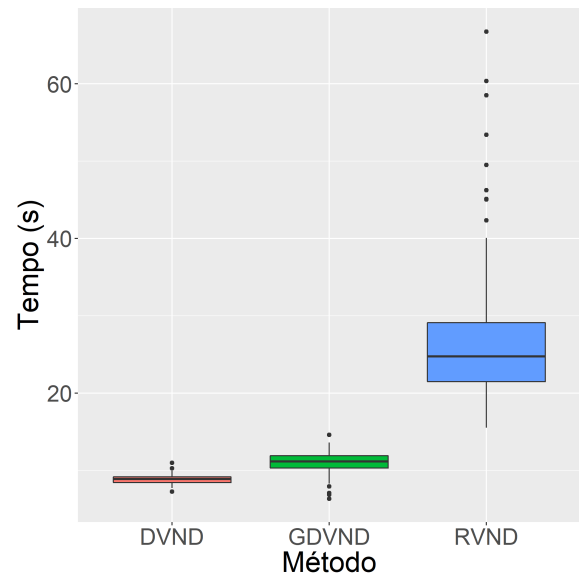
gasto ao combinar os movimentos é grande contudo o DVND ainda consegue alcançar melhores tempos em algumas das instâncias.

Finalmente, nas instâncias 6 e 7 (Figuras 42c e 42d) a diferença se mostra bastante significativa e o tempo do *GDVND-MAN* é menor na maioria das amostras.

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

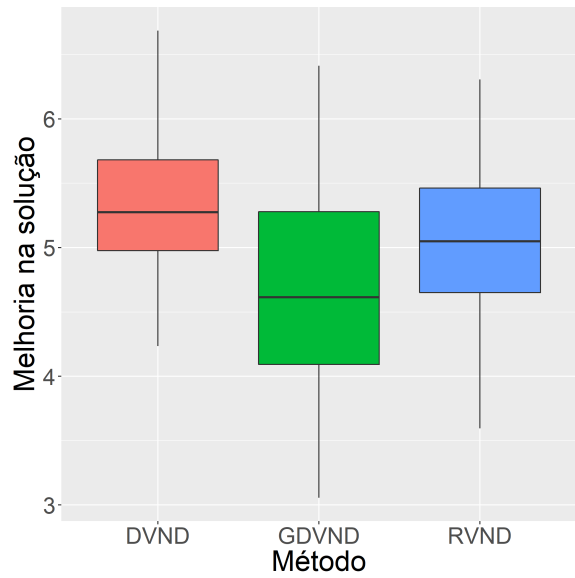
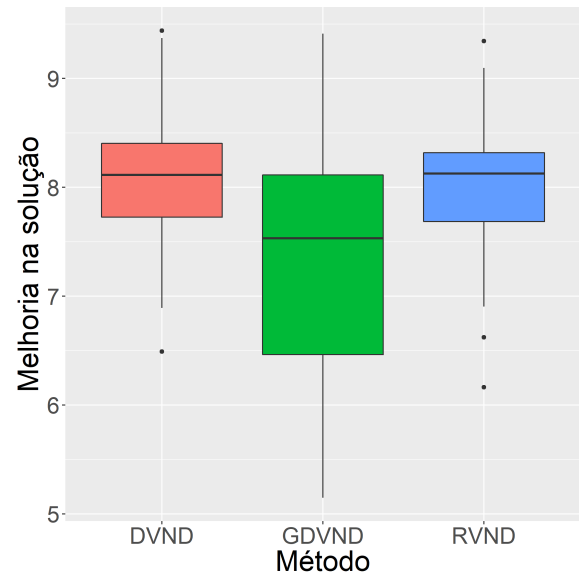
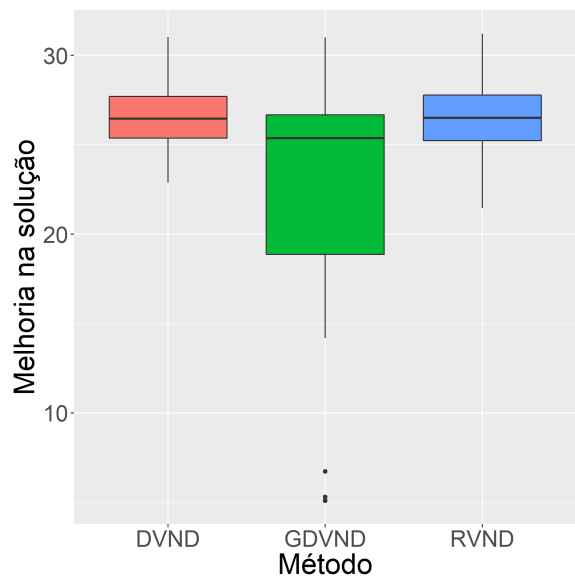
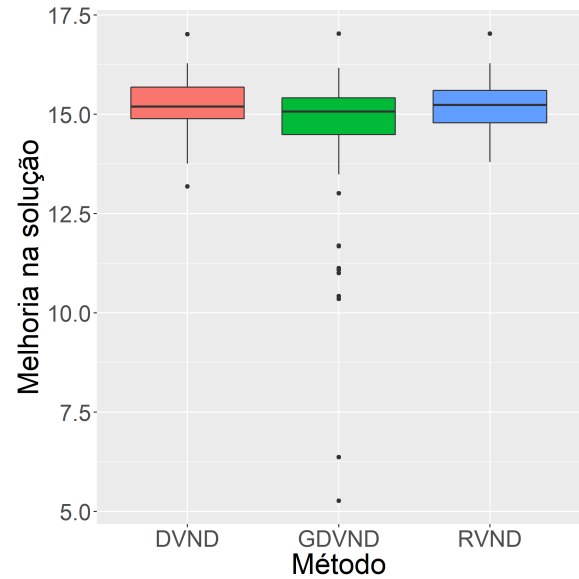
Método ■ DVND ■ GDVND ■ RVND

Figura 37 – Tempo dos algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 0 a 3.

(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

Método  DVND  GDVND  RVND

Figura 38 – Tempo dos algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 4 a 7.

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

Método  DVND  GDVND  RVND

Figura 39 – Melhoria no valor da solução para os algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 0 a 3.

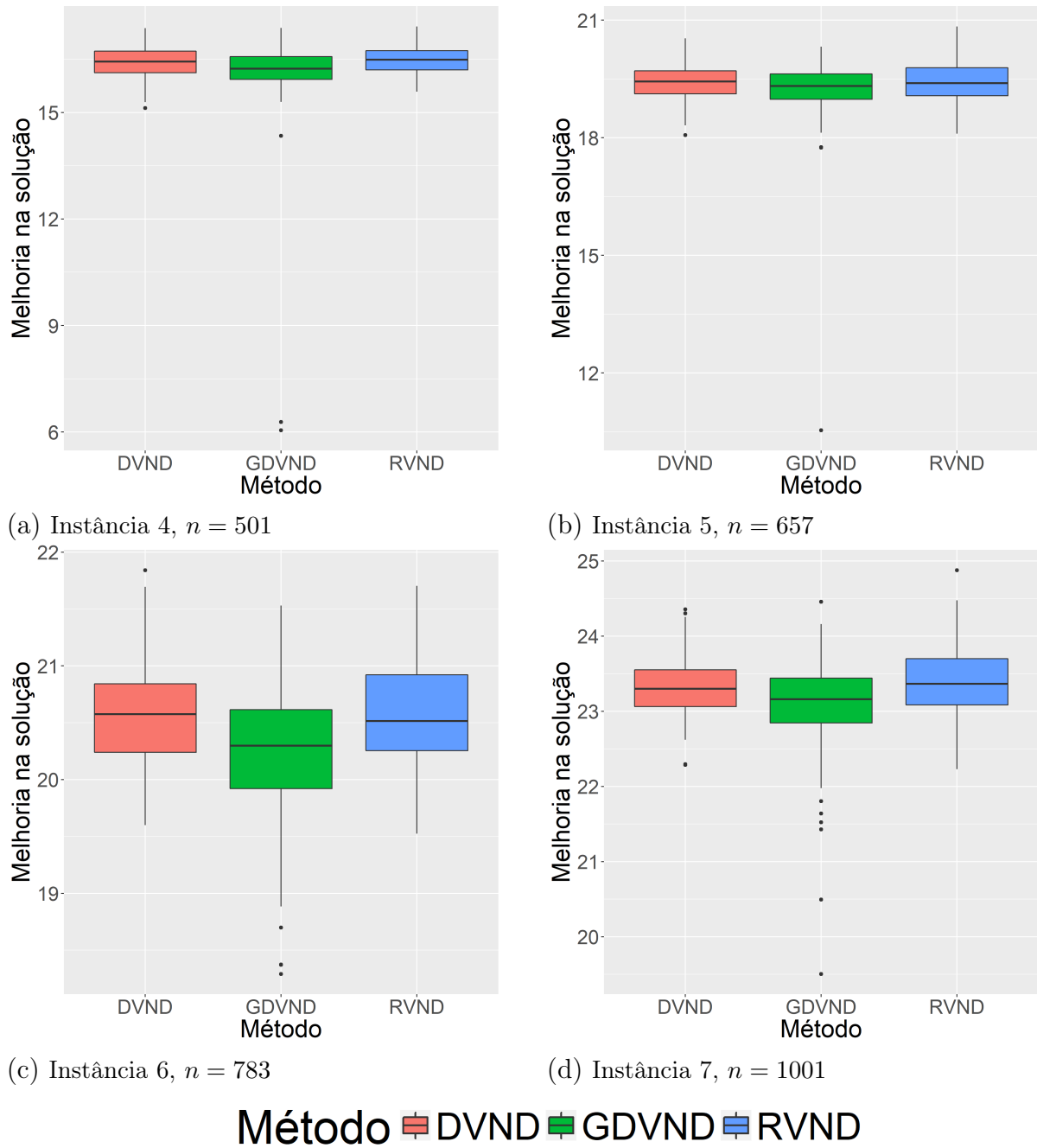
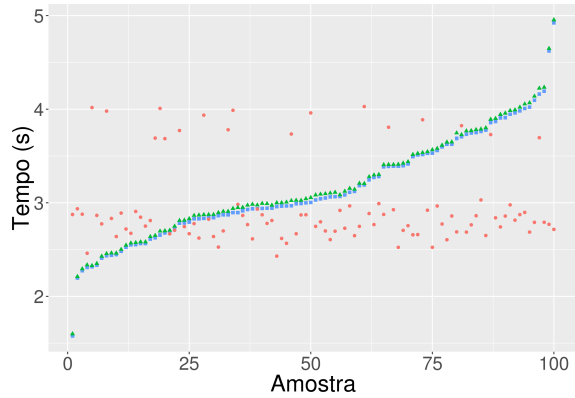
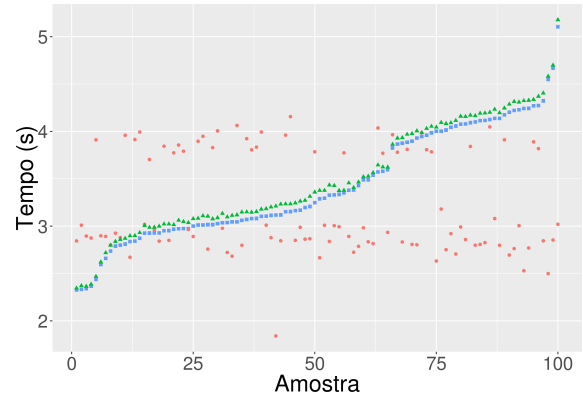
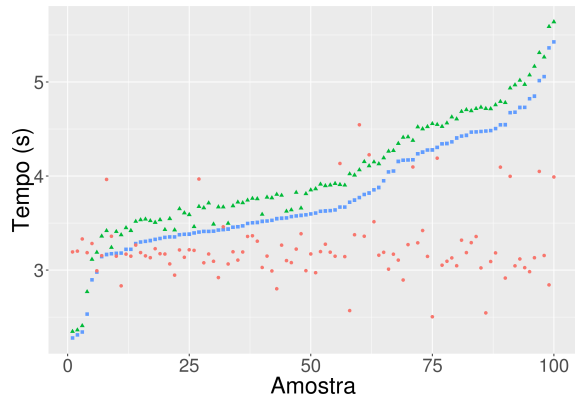
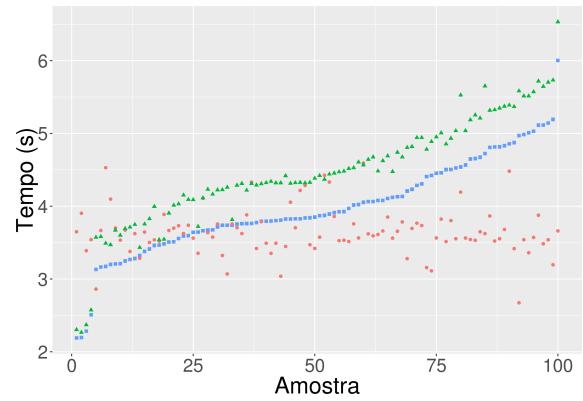
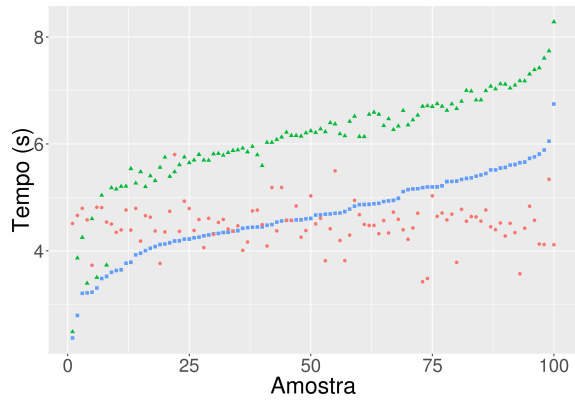
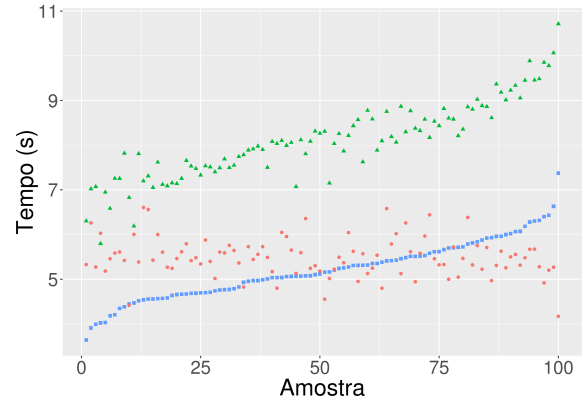
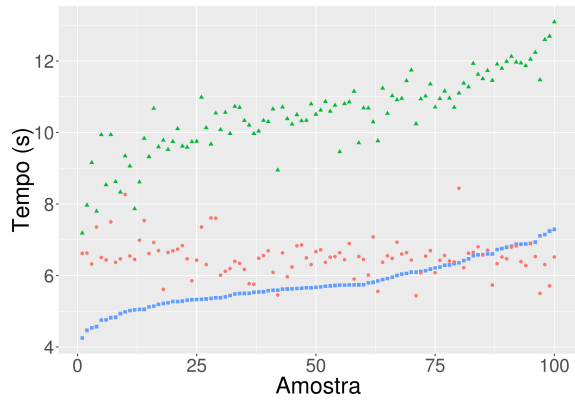
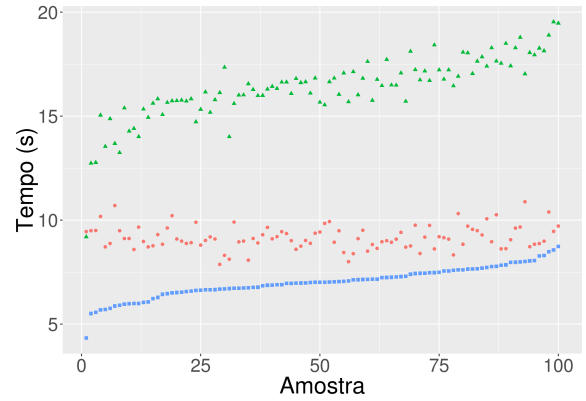


Figura 40 – Melhoria no valor da solução para os algoritmos GDVND, DVND e RVND, n representa o tamanho da instância. Instâncias 4 a 7.

(a) Instância 0, $n = 52$ (b) Instância 1, $n = 100$ (c) Instância 2, $n = 226$ (d) Instância 3, $n = 318$

Método ● DVND ▲ GDVND ■ GDVND-MAN

Figura 41 – Tempo do DVND vs GDND, *DVND* refere-se ao tempo gasto pelo algoritmo de mesmo nome, para *GDVND* é análogo ao anterior, no caso do *GDVND-MAN* este se refere ao tempo do *GDVND* subtraído do tempo para gerenciar os movimentos, n representa o tamanho da instância, m indica o número de máquinas. Instâncias 0 a 3.

(a) Instância 4, $n = 501$ (b) Instância 5, $n = 657$ (c) Instância 6, $n = 783$ (d) Instância 7, $n = 1001$

Método ● DVND ▲ GDVND ■ GDVND-MAN

Figura 42 – Tempo do DVND vs GDND, *DVND* refere-se ao tempo gasto pelo algoritmo de mesmo nome, para *GDVND* é análogo ao anterior, no caso do *GDVND-MAN* este se refere ao tempo do *GDVND* subtraído do tempo para gerenciar os movimentos, n representa o tamanho da instância, m indica o número de máquinas. Instâncias 4 a 7.

5 CONCLUSÕES

Foi possível simular uma memória global em um dataflow pelo uso do nó de *flip-flop* apresentado na Seção 3.2.2 de forma a prover uma memória para o nó *man* do grafo dataflow do DVND, expresso na Figura 18.

Conforme discutido na Seção 3.2.3, foi proposta a melhoria da biblioteca *Sucuri* para que os nós de seus grafos comportem múltiplas portas de saída, sendo visto na Seção 4.3 que os resultados indicam sua eficiência para instâncias com tamanho maior ou igual a 318.

5.1 RVND

Foi discutido na Seção 4.4 que em termos de melhoria na qualidade da solução não foi encontrada grande diferença nos métodos RVND em sua implementação clássica e da implementação em dataflow.

Em termos de tempo de execução houve uma pequena diferença em algumas instâncias pesando para a implementação dataflow e em outras para a implementação clássica.

5.2 DVND

No que diz respeito a comparação entre o DVND implementado em dataflow e a implementação original podemos ver que a versão clássica apresentou melhores tempos para as menores instâncias, sendo alcançado pela tempo da implementação em dataflow apenas na instância 5, de tamanho 657. Apenas na instância 7 (tamanho 1001) o DVND em dataflow conseguiu melhorar o tempo do DVND clássico, contudo a melhoria dos resultados com o aumento do tamanho da solução indica que o mesmo possui tempo de execução mais controlado para grandes instâncias.

Em termos de valor da solução encontrada pode se ver na discussão da Seção 4.5 que o DVND clássico conseguiu melhorar mais a solução inicial quando comparado à implementação em dataflow.

É importante ressaltar que tanto a implementação clássica quanto a implementação em dataflow do DVND melhoraram o tempo de execução quando comparadas ao RVND dataflow ou mesmo o clássico.

5.3 GDVND

Desta forma foi possível mostrar que o GDVND consegue diminuir a necessidade de explorar vizinhanças pois, conforme discutido na Seção 4.6.3, o tempo gasto por esse na exploração de vizinhanças é menor para as maiores instâncias, contudo carece de uma melhor estratégia para combinar os movimentos pois este está tomando uma parte significativa do tempo de execução do procedimento como um todo.

5.4 Propostas futuras

No desenvolvimento da experimentação e da análise dos resultados algumas novas hipóteses foram levantadas e são aqui apresentadas para estudo posterior.

5.4.1 Testar com instâncias maiores

No intuito de verificar melhor o desempenho do método dataflow DVND e sua escalabilidade uma prova de conceito imaginada pra ser utilizada é a realização de teste computacionais para instâncias maiores, tendo em vista a comparação de resultados com instâncias do clássico PCV.

5.4.2 Decomposição de vizinhanças

Conforme descrito na seção 3.5, as vizinhanças exploradas nos problemas não são indivisíveis, desta forma uma maneira de proporcionar maior paralelismo pode ser feita através da decomposição das destas em sub vizinhanças de forma a serem exploradas paralelamente.

Acredita-se que a decomposição de vizinhanças aliada a composição de movimentos pode proporcionar um grande ganho em termos de qualidade da solução uma convergência muito mas rápida ao serem aplicados mais de um movimento simultaneamente, melhorando assim o tempo da busca local que em geral é a etapa mais custosa em termos computacionais para o processo de solução de um problema de otimização.

REFERÊNCIAS

ALVES, Tiago et al. A minimalistic dataflow programming library for python. In: IEEE. Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on. [S.l.], 2014.

ALVES, Tiago A.O. et al. Trebuchet: exploring TLP with dataflow virtualisation. International Journal of High Performance Systems Architecture, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 3, n. 2/3, p. 137, 2011. ISSN 1751-6528.

ALVES, Tiago A.O.; MARZULO, Leandro A. J.; FRANCA, Felipe M. G. Unleashing parallelism in longest common subsequence using dataflow. In: 4th Workshop on Applications for Multi-Core Architectures. [S.l.: s.n.], 2013.

ARAUJO, Rodolfo Pereira; COELHO, Igor Machado; MARZULO, Leandro A. J. A dvnd local search implemented on a dataflow architecture for the minimum latency problem. In: 7th Workshop on Parallel Programming Models (32nd IEEE International Parallel and Distributed Processing Symposium - IPDPS 2018). [S.l.: s.n.], 2018.

ARAUJO, Rodolfo Pereira et al. A novel list-constrained randomized vnd approach in gpu for the traveling thief problem. Eletronic Notes in Discrete Mathematics, v. 66, p. 183–190, 2018.

BALAKRISHNAN, Saisanthosh; SOHI, G.S. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. In: 33rd International Symposium on Computer Architecture (ISCA'06). Washington, DC, USA: IEEE, 2006. p. 302–313. ISBN 0-7695-2608-X.

BIANCO, Lucio; MINGOZZI, Aristide; RICCIARDELLI, Salvatore. The traveling salesman problem with cumulative costs. Networks, Wiley Online Library, v. 23, n. 2, p. 81–91, 1993.

BOSILCA, George et al. Dague: A generic distributed dag engine for high performance computing. Parallel Computing, v. 38, n. 1-2, p. 37–51, 2012.

DAVIS, A. L. The architecture and system method of ddm1: A recursively structured data driven machine. In: Proceedings of the 5th Annual Symposium on Computer Architecture. New York, NY, USA: ACM, 1978. (ISCA '78), p. 210–215. Disponível em: <<http://doi.acm.org/10.1145/800094.803050>>.

DENNIS, Jack B.; MISUNAS, David P. A preliminary architecture for a basic data-flow processor. SIGARCH Comput. Archit. News, ACM Press, New York, NY, USA, v. 3, n. 4, p. 126–132, 1974. ISSN 0163-5964.

DURAN, Alejandro et al. Ompss: A proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters, v. 21, p. 173–193, 2011-03-01 2011.

FLYNN, Michael J. Some computer organizations and their effectiveness. IEEE Transactions on Computers, IEEE, C-21, n. 9, p. 948–960, set. 1972.

GAREY, Michael R.; JOHNSON, David S. Computers and Intractability; A Guide to the Theory of NP-Completeness. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN 0716710455.

GENDREAU, Michel; POTVIN, Jean-Yves (Ed.). Handbook of Metaheuristics. Springer, 2010. Disponível em: <<https://EconPapers.repec.org/RePEc:spr:isorms:978-1-4419-1665-5>>.

GIORGI, Roberto et al. TERAFLUX: Harnessing dataflow in next generation teradevices. Microprocessors and Microsystems, Elsevier, p. –, 2014. Available online 18 April 2014.

GLOVER, Fred W; KOCHENBERGER, Gary A. Handbook of metaheuristics. [S.l.]: Springer Science & Business Media, 2006. v. 57.

GRAFE, V. G. et al. The epsilon dataflow processor. In: Proceedings of the 16th Annual International Symposium on Computer Architecture. New York, NY, USA: ACM, 1989. (ISCA '89), p. 36–45. ISBN 0-89791-319-1. Disponível em: <<http://doi.acm.org/10.1145/74925.74930>>.

GUPTA, Gagan; SOHI, Gurindar S. Dataflow execution of sequential imperative programs on multicore architectures. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA: ACM, 2011. (MICRO-44), p. 59–70. ISBN 978-1-4503-1053-6. Disponível em: <<http://doi.acm.org/10.1145/2155620.2155628>>.

GURD, J R; KIRKHAM, C C; WATSON, I. The Manchester prototype dataflow computer. Communications of the ACM, ACM, New York, NY, USA, v. 28, n. 1, p. 34–52, jan. 1985. ISSN 00010782.

HENNESSY, John L; PATTERSON, David A. Computer architecture: a quantitative approach. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. 856 p. ISBN 9780123838728.

KISHI, Masasuke; YASUHARA, Hiroshi; KAWAMURA, Yasusuke. Dddp-a distributed data driven processor. SIGARCH Comput. Archit. News, ACM, New York, NY, USA, v. 11, n. 3, p. 236–242, jun. 1983. ISSN 0163-5964. Disponível em: <<http://dl.acm.org/citation.cfm?id=1067651.801661>>.

MARZULO, Leandro Augusto Justen. Explorando linhas de execução paralelas com programação orientada por fluxo de dados. Tese (Doutorado) — Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia, Rio de Janeiro, RJ, Brasil, 10 2011. Universidade Federal do Rio de Janeiro.

MLADENOVIC, Nenad; HANSEN, Pierre. Variable neighborhood search. Computers & Operations Research, Elsevier, v. 24, n. 11, p. 1097–1100, nov. 1997.

MLADENOVIC, Nenad; UROSEVIC, Dragan; HANAFI, Saïd. Variable neighborhood search for the travelling deliveryman problem. 4OR, Springer, v. 11, n. 1, p. 57–73, 2013.

NVIDIA. CUDA Pro Tip: Occupancy API Simplifies Launch Configuration. 2014. Disponível em: <<https://devblogs.nvidia.com/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/>>.

PAPADOPOULOS, G. M.; CULLER, D. E. Monsoon: an explicit token-store architecture. In: [1990] Proceedings. The 17th Annual International Symposium on Computer Architecture. [S.l.: s.n.], 1990. p. 82–91.

PATTERSON, David; HENNESSY, John. Computer organization and design (3rd ed.): the hardware/software interface. Morgan Kaufmann Publishers Inc., 2003. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 1558604286. Disponível em: <<https://www.elsevier.com/books/computer-organization-and-design/patterson/978-0-08-050257-1>>.

RIOS, Eyder et al. A benchmark on multi improvement neighborhood search strategies in cpu/gpu systems. In: WAMCA 2016 (SBAC 2016). [S.l.: s.n.], 2016.

_____. A performance study on gpu-based neighborhood search algorithms for vehicle routing. In: 6th Workshop on Applications for Multi-core Architectures (WAMCA), held in conjunction with SBAC-PAD'2015. Florianópolis, SC, Brazil: [s.n.], 2015.

_____. Exploring parallel multi-gpu local search strategies in a metaheuristic framework. JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING, v. 111, p. 39–55, 2017.

_____. A performance study on multi improvement neighborhood search strategy. ELECTRONIC NOTES IN DISCRETE MATHEMATICS, v. 58, p. 199–206, 2017.

_____. Exploring parallel multi-gpu local search strategies in a metaheuristic framework. Journal of Parallel and Distributed Computing, v. 111, p. 39 – 55, 2018. ISSN 0743-7315. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0743731517302009>>.

SAKAI, S. et al. An architecture of a dataflow single chip processor. In: The 16th Annual International Symposium on Computer Architecture. [S.l.: s.n.], 1989. p. 46–53. ISSN 1063-6897.

SHIMADA, T. et al. Evaluation of a prototype data flow processor of the sigma-1 for scientific computations. SIGARCH Comput. Archit. News, ACM, New York, NY, USA, v. 14, n. 2, p. 226–234, maio 1986. ISSN 0163-5964. Disponível em: <<http://doi.acm.org/10.1145/17356.17383>>.

SILVA, Marcos Melo et al. A simple and effective metaheuristic for the minimum latency problem. EJOR, Elsevier B.V., v. 221, n. 3, p. 513–520, Sep 2012. ISSN 03772217. Disponível em: <<http://linkinghub.elsevier.com/retrieve/pii/S037722171200269X>>.

SILVA, Rafael J.N. et al. Task Scheduling in Sucuri Dataflow Library. In: 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). IEEE, 2016. v. 1, p. 37–42. ISBN 978-1-5090-4844-1. Disponível em: <<http://ieeexplore.ieee.org/document/7803693/>>.

SimplePyCuda. 2017. <<https://github.com/igormcoelho/simple-pycuda>>. Accessed: 2018-02-16.

SOUZA, M.J.F. et al. A hybrid heuristic algorithm for the open-pit-mining operational planning problem. European Journal of Operational Research, v. 207, n. 2, p. 1041 – 1051, 2010. ISSN 0377-2217. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0377221710003875>>.

SUBRAMANIAN, Anand; UCHOA, Eduardo; OCHI, Luiz Satoru. A hybrid algorithm for a class of vehicle routing problems. Computers & Operations Research, Elsevier, v. 40, n. 10, p. 2519–2531, out. 2013.

SWANSON, S et al. WaveScalar. In: Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on. [S.l.]: IEEE Comput. Soc, 2003. p. 291–302. ISBN 0-7695-2043-X.

SWANSON, Steven et al. The wavescalar architecture. ACM Trans. Comput. Syst., ACM, New York, NY, USA, v. 25, n. 2, p. 4:1–4:54, maio 2007. ISSN 0734-2071. Disponível em: <<http://doi.acm.org/10.1145/1233307.1233308>>.

TBB FlowGraph. accessed on August 8, 2014. Disponível em: <http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm>.

TOMASULO, R M. An efficient algorithm for exploring multiple arithmetic units. IBM Journal of Research and Development, v. 11, p. 25–33, jan. 1967.

TSITSIKLIS, John N. Special cases of traveling salesman and repairman problems with time windows. Networks, v. 22, n. 3, p. 263–282, maio 1992.