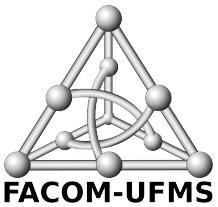
Metaheurísticas para o Problema da Mochila Multidimensional

Bianca de Almeida Dantas

Tese de Doutorado

Orientação
Prof. Dr. Edson Norberto Cáceres



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Campo Grande - MS
2016

Aos meus pais Adelina e Felix.
Ao meu namorado Silvio.
Ao meu irmão Cleitor.

Agradecimentos

Agradeço, em primeiro lugar, a Deus por me guiar em todos os momentos de minha vida e me fortalecer diante das dificuldades.

À minha mãe, eterna companheira e melhor amiga Adelina, que sempre foi a minha fortaleza e que, infelizmente, não pôde acompanhar fisicamente esta etapa de minha vida até o final, mas continua a me inspirar em todos os momentos. Ao meu pai Felix, pela amizade, carinho e ombro amigo sempre presentes quando precisei. Agradeço pelo amor, exemplo e motivação que meus pais me deram para alcançar meus objetivos desde os meus primeiros passos.

Ao meu namorado e grande amigo Silvio, presença e apoio constantes em todos os momentos de desenvolvimento deste trabalho, ouvindo minhas inúmeras divagações, assistindo e cronometrando as prévias de minhas apresentações e dando opiniões e dicas nas horas certas. Agradeço pela paciência em aguentar minhas ausências e pelo incentivo na minha vida.

Ao meu orientador Edson Norberto Cáceres, por sempre acreditar em meu potencial desde os tempos da graduação. Agradeço por estar sempre disponível para sanar as minhas dúvidas e me guiar na minha carreira acadêmica.

Ao meu querido irmão Cleitor, sempre pronto para me ajudar nas minhas milhares de instalações e configurações do Linux. Agradeço a todos da minha família pelo apoio, em especial, minhas tias Edilce e Edenide, meus tios Alício e Cecílio, minhas primas Laís, Ivana e Silvana, minha cunhada Michele, minha sobrinha Diuly e dona Elaine.

Ao casal de amigos e colegas de Doutorado e de trabalho Jucele e Francisco, pela ajuda nas dificuldades e pelos muitos estudos e trabalhos realizados desde os tempos do IFMS. Agradeço às amigas e colegas de sala Valéria, Raquel e Daniela pelas longas conversas.

À professora Edna Ayako Hoshino pela troca de ideias e pelas diversas sugestões ao longo de minhas pesquisas. Agradeço a João Gabriel Felipe Machado Gazolla pelas dicas no uso da memória de textura.

Aos professores Nalvo Franco de Almeida Junior e Marcelo Henriques de Carvalho, respectivamente, diretor da Faculdade de Computação e coordenador do curso de Doutorado em Ciência da Computação, e a todos os funcionários da FACOM pelo apoio no desenvolvimento deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Resumo

O problema da mochila multidimensional é amplamente conhecido na área de otimização combinatória e possui diversas aplicações práticas, tais como, dimensionamento de cargas, orçamento de capital, alocação de recursos em sistemas distribuídos, entre outros. Apesar de sua popularidade, sua solução não é trivial, de fato, ele pertence a uma classe de problemas conhecidos como $\mathcal{NP}$-difíceis, para os quais não são conhecidos algoritmos polinomiais capazes de obter uma solução exata para todas as instâncias. Essa situação motiva a busca por técnicas alternativas para obter soluções em menor tempo, ainda que aproximadas e, nesse cenário, as metaheurísticas têm se mostrado de especial interesse, pois são capazes de alcançar soluções de boa qualidade para uma ampla variedade de problemas. Entretanto, mesmo as metaheurísticas podem ser relativamente demoradas, em especial para instâncias de tamanho grande, o que encoraja a aplicação de técnicas, como as estratégias de paralelização, para reduzir os seus tempos de execução ou, ainda, melhorar a qualidade das soluções. Neste trabalho, apresenta-se o estudo, implementação e análise de um conjunto de metaheurísticas para a solução do problema da mochila multidimensional. São apresentadas e avaliadas diferentes alternativas para a paralelização e os resultados obtidos são comparados com os de outros trabalhos da literatura pesquisada. Para comprovar que os bons resultados possibilitados pelo uso das metaheurísticas estudadas não foram obtidos ao acaso, as soluções foram validadas com a utilização de testes estatísticos.

Palavras-chave: *Problema da Mochila Multidimensional*, *Metaheurísticas*, *Programação Paralela*.

Abstract

The 0-1 multidimensional knapsack problem (0-1 MKP) is widely known in combinatorial optimization with numerous practical applications, such as cargo loading, capital budgeting, resource allocation in distributed systems, among others. Despite its popularity, it is not a trivial problem, in fact, 0-1 MKP belongs to $\mathcal{NP}$-hard class of problems, for which there are not polynomial-time algorithms capable of obtaining exact solutions for every possible instance. This situation motivates the search for alternative techniques aiming to achieve solutions in reduced execution time, albeit approximate and, in this case, metaheuristics have become specially attractive since they can lead to good quality solutions to a wide range of problems. However, even metaheuristics can be relatively slow, specially with large size instances, which encourages the application of techniques, such as parallelization strategies, aiming to reduce execution times or even improve the quality of solutions. In this work, we present the study, implementation and analysis of a set of metaheuristics for the solution of the 0-1 multidimensional knapsack problem. Different approaches for parallelization are presented and evaluated and the achieved results are compared to other works. Achieved solutions were validated with application of statistical tests, in order to show that the good quality solutions obtained by the metaheuristics were not obtained by chance.

Keywords: *0-1 Multidimensional Knapsack Problem*, *Metaheuristics*, *Parallel Programming.*

CAPÍTULO 1

Introdução

O problema da mochila é um problema clássico de otimização combinatória que possui uma ampla gama de aplicações e cuja variante 0-1, a mais conhecida, pode ser vista como um problema de programação inteira com uma única restrição no qual, a princípio, qualquer outro problema desse tipo pode ser transformado. Dessa forma, uma solução eficiente para o problema é de interesse para toda a área da programação inteira. O problema da mochila 0-1, ou problema da mochila binária, possui basicamente duas entradas: um conjunto de itens (com seus valores e pesos associados) e a capacidade da mochila; nesta variante há apenas duas possibilidades para cada item: ele deve ou não ser levado. Apesar de sua relativa simplicidade, esse problema é $\mathcal{NP}$-difícil, ou seja, não é conhecido um algoritmo polinomial para sua solução exata. Existem basicamente duas abordagens para encontrar a solução exata para este problema, programação dinâmica e *branch & bound*, cuja adequação depende de características da instância do problema a ser resolvido. O algoritmo de programação dinâmica utilizado para solução do problema possui complexidade $O(nW)$ e, pelo fato de depender tanto do número de itens (n) quanto da capacidade (W) da mochila, é denominado *pseudopolinomial*.

Quando o número de restrições impostas ao problema da mochila aumenta, tem-se o chamado problema da mochila multidimensional e fica evidente que as alternativas de solução tradicional não podem mais ser aplicadas de maneira eficiente. Essa situação motiva a busca por alternativas para obter uma solução, ainda que aproximada, para o problema dado que a obtenção da solução exata pode ser proibitiva em alguns casos, devido ao seu alto custo. É nesse contexto que a utilização de algoritmos de aproximação e heurísticas surge como uma opção.

Um algoritmo de aproximação é um algoritmo que soluciona um determinado problema respeitando a restrição de que a solução alcançada não possui qualidade inferior a uma fração da solução ótima, ou seja, existe uma garantia a respeito do quão grande é a perda da qualidade ao se fazer uso desse tipo de algoritmo [24]. Algoritmos de aproximação foram usados para solucionar o problema da mochila binário, Sahni [67] propôs um esquema de aproximação de tempo polinomial (*polinomial time approximation scheme - PTAS*) e Ibarra e Kim [38] um esquema de aproximação de tempo completamente polinomial (*fully polinomial time approximation scheme - FPTAS*).

Posteriormente, Lawler [44] propôs alterações ao algoritmo de Ibarra e Kim para levar a melhores limites de tempo e espaço. Tendo em mente as implementações anteriores, Magazine e Oguz [48] propuseram uma alteração baseada em um algoritmo de programação dinâmica para chegar à solução e Kellerer e Pferschy [39] apresentaram uma nova versão de FPTAS que reduziu os requisitos de espaço. Caprara *et al.* [15] descreveram algoritmos PTAS e FPTAS para uma variante do problema clássico, o problema da mochila de k itens, na qual uma constante k limita superiormente o número de itens da solução. Frieze e Clarke [30], por sua vez, propuseram um esquema de aproximação polinomial para o problema da mochila multidimensional baseado no uso do algoritmo simplex dual para programação linear.

As heurísticas são técnicas que, em geral, são capazes de obter boas soluções para diversos problemas, entretanto, diferentemente dos algoritmos de aproximação, não há garantia alguma sobre a qualidade de tais soluções. Apesar de serem alternativas viáveis para resolver problemas difíceis, mesmo heurísticas podem ser bastante custosas, motivando a utilização de técnicas para redução do tempo de execução ou para melhorar a qualidade das soluções como, por exemplo, estratégias de paralelização.

O objetivo deste trabalho é apresentar um estudo comparativo de diferentes metaheurísticas para a solução do problema da mochila multidimensional. Para tal, as metaheurísticas foram implementadas usando abordagens sequenciais e paralelas e testadas com os mesmos conjuntos de testes usados em trabalhos de referência. Com base nos resultados alcançados, validados por meio da aplicação de testes estatísticos, pôde-se comprovar a viabilidade do uso de tais técnicas e que a utilização de estratégias de paralelização pode levar a resultados de melhor qualidade e/ou a menores tempos de execução.

As implementações paralelas foram realizadas com o uso de GPGPU (*general purpose computing on graphics processing units*). No caso dos algoritmos genéticos e das redes neurais aumentadas, que foram as duas primeiras heurísticas estudadas no desenvolvimento deste trabalho, também foram desenvolvidas versões em MPI com o uso de um *cluster*. O intuito no uso dessas duas estratégias paralelas foi avaliar o quanto as heurísticas poderiam ser beneficiadas com o uso de múltiplos núcleos de processamento, bem como qual das estratégias seria mais vantajosa. Para obter os tempos de execução equivalentes aos alcançados com o uso de GPGPU, os programas em MPI demandaram o aumento do número de processadores, entretanto, tal aumento implicava na redução da qualidade das soluções obtidas. Essa situação desmotivou o uso de MPI, o que, aliado ao fato de que o acesso a placas de vídeo com centenas a milhares de núcleos é mais fácil do que a *clusters*, motivou a decisão de concentrar os esforços de paralelização no uso de GPGPU e, assim, as demais heurísticas não foram mais implementadas usando MPI.

O restante do texto está organizado como segue. O Capítulo 2 apresenta a definição do problema da mochila multidimensional e de metaheurísticas e descreve o ambiente de testes e o processo de validação dos resultados alcançados. As metaheurísticas estudadas são os temas dos Capítulos 3, 4, 5, 6 e 7, os quais abordam, respectivamente, algoritmos genéticos, redes neurais aumentadas, GRASP, *simulated annealing* e a abordagem híbrida proposta de redes neurais aumentadas e GRASP; esses capítulos descrevem as fundamentações teóricas, as implementações e os resultados obtidos para os conjuntos de testes. O Capítulo 8 apresenta uma análise comparativa das me-

taheurísticas estudadas. Por fim, o Capítulo 9 discorre sobre as conclusões do trabalho e sugestões para trabalhos futuros.

CAPÍTULO 2

Contextualização

2.1 Problema da Mochila Multidimensional

Um dos problemas mais conhecidos da área de otimização combinatória é o *problema da mochila*. Neste problema, tem-se uma mochila com uma determinada capacidade máxima e um conjunto de itens – cada um com um peso e um valor associado –, deve-se escolher quais itens levar na mochila de forma a maximizar o valor a ser carregado sem extrapolar a capacidade da mochila.

Existem diversas variantes do problema na literatura, as quais se diferenciam, em geral, pelo número de restrições impostas à solução ou se os itens são ou não particionáveis, dentre elas:

- Mochila fracionária: Os itens podem ser particionados e pode-se colocar frações de cada item na mochila;
- Mochila 0-1 (mochila binária): os itens são indivisíveis e podem ou não estar presentes na solução;
- Mochila multidimensional: extensão da mochila binária, considerando mais de uma restrição;
- Mochila multiobjetivo: extensão da mochila multidimensional que considera mais de um objetivo a ser maximizado, assim, cada item tem mais de um valor associado, um por objetivo.

Este trabalho tem como foco principal o problema da mochila multidimensional, entretanto, para sua melhor compreensão é necessário, inicialmente, compreender a versão binária. O problema da mochila 0-1 pode ser formalmente definido como [13]: dados um conjunto de n diferentes itens $S = \{1, 2, ..., n\}$, os conjuntos de seus valores e de seus pesos – representados, respectivamente, pelos vetores $V = \{v_1, v_2, ..., v_n\}$ e $P = \{p_1, p_2, ..., p_n\}$ – e uma mochila com capacidade W, deve-se definir quais itens

maximizam o valor a ser carregado na mochila sem exceder a sua capacidade, ou seja:

$$\begin{aligned} &\max\{\sum_{i=1}^{n} v_i x_i\} \\ &\text{tal que:} \sum_{i=1}^{n} p_i x_i \leq W, x_i \in \{0,1\} \end{aligned} \tag{2.1}$$

onde $X = \{x_1, x_2, ..., x_n\}$ é o chamado vetor de solução, que especifica se cada item i está ou não presente na solução.

Este é um problema de programação inteira com uma única restrição no qual, a princípio, qualquer outro problema de programação inteira pode ser transformado e que pertence à classe dos problemas $\mathcal{NP}$-difíceis. O problema pode ser solucionado sequencialmente em tempo $O(nW)$, o que configura uma solução *pseudopolinomial*, dado que, além do número de itens, a complexidade também depende da capacidade da mochila. O algoritmo que implementa a solução pseudopolinomial para o problema da mochila 0-1 utiliza a estratégia de *programação dinâmica* e foi primeiramente proposto por Gilmore e Gomory [31] em 1966.

O problema da mochila multidimensional pode ser visto como uma extensão da mochila binária. Nesta versão, o conjunto de itens da solução deve considerar um conjunto de recursos, ao invés de apenas a capacidade da mochila. Pode-se formular o problema de acordo com a Equação 2.2 [17]:

$$\begin{aligned} &\max\{\sum_{j=1}^{n} v_j x_j\} \\ &\text{tal que:} \sum_{j=1}^{n} r_{ij} x_j \leq b_i, i = 1, \ldots, m, \\ &x_j \in \{0,1\}, j = 1, \ldots, n. \end{aligned} \tag{2.2}$$

onde n é o número de itens, m representa a quantidade de recursos, $V = \{v_1, v_2, \ldots, v_n\}$ é o vetor de valores dos itens, $B = \{b_1, b_2, \ldots, b_m\}$ é o vetor de capacidades dos recursos e $X = \{x_1, x_2, \ldots, x_n\}$ é o vetor de solução, no qual cada elemento pode possuir os valores 1 ou 0, equivalendo, respectivamente, à sua presença ou não na solução. A matriz $R = \{r_{11}, \ldots, r_{1n}, \ldots, r_{m1}, \ldots, r_{mn}\}$, por sua vez, representa o quanto cada item usa de cada recurso. Assim como na versão binária do problema, a solução do problema consiste em encontrar o vetor solução X que satisfaça as restrições do problema.

Uma alternativa para obter uma solução ótima para o problema é aplicar programação dinâmica considerando, agora, os m recursos. É imediato perceber que a complexidade dessa alternativa é significantemente maior do que no caso anterior, uma vez que será necessário construir uma matriz de programação dinâmica de $m + 1$ dimensões – diferentemente das duas dimensões da mochila binária. Essa característica torna proibitivo o uso da programação dinâmica para instâncias grandes do problema; de fato, a obtenção de uma solução exata para a mochila multidimensional é um problema $\mathcal{NP}$-difícil e, em geral, não é aconselhada.

Como a obtenção da solução exata para o problema é um processo custoso, o uso de técnicas que buscam por soluções aproximadas tem sido intensivamente pesquisadas como, por exemplo, as metaheurísticas estudadas nos próximos capítulos.

2.2 Metaheurísticas

Uma metaheurística é um conjunto de conceitos usados para a definição de métodos heurísticos que podem ser aplicados a diferentes problemas de otimização. Em geral, são necessárias poucas modificações para aplicar metaheurísticas a diferentes problemas e as soluções obtidas tendem a ser de boa qualidade [10]. Um grande número de trabalhos dedicados à aplicação de metaheurísticas para resolver problemas de otimização combinatória têm mostrado a eficácia de tais métodos, motivando o estudo de suas diferentes variantes na solução de problemas para os quais a obtenção de uma solução exata é difícil, em especial para a classe de problemas $\mathcal{NP}$-difíceis, como o problema da mochila multidimensional, objeto das análises e implementações deste trabalho.

No que diz respeito ao problema da mochila multidimensional, algumas das metaheurísticas mais comumente utilizadas para a sua solução são os algoritmos genéticos, *simulated annealing*, busca tabu e redes neurais [70]. Niar e Freville [54] propuseram uma metaheurística paralela baseada em busca tabu e mostraram que a aplicação de paralalelismo possibilitou a configuração automática e dinâmica de alguns parâmetros da busca, o que levou à diminuição do tempo necessário para obter uma solução. Algoritmos genéticos (AGs) foram explorados por um grande número de autores como, por exemplo, Khuri, Bläck e Heitkötter [40] e Hoff, Løkkentangen e Mittet [36], merecendo destaque a implementação proposta por Chu e Beasley [17]. Chu e Beasley consideraram informações específicas do problema para melhorar a qualidade dos resultados obtidos com relação a trabalhos anteriores. Além disso, eles foram responsáveis por gerar um conjunto de instâncias, maiores que as previamente conhecidas, com as quais mostraram que o algoritmo era eficiente para instâncias maiores que as usadas até aquele momento. Uma versão paralela de algoritmos genéticos foi proposta por Posadas *et al.* [58] para resolver instâncias de tamanho médio; os autores usaram as bibliotecas GALib [1] e OOMPI [2] para implementar as funcionalidades de AGs e de comunicação. A otimização por colônia de formigas foi utilizada por Liu e Lv [47] e por Fingler *et al.* [28]; os primeiros autores descreveram um algoritmo paralelo utilizando o modelo de programação *map-reduce* que foi executado com grandes instâncias de teste usando computação em nuvem. Os autores do segundo trabalho desenvolveram uma versão paralela, usando GPGPU com CUDA, da estratégia proposta por Solnon e Bridge [69], a implementação proposta obteve bons resultados com tempos de execução reduzidos. Deane e Agarwal descreveram uma rede neural aumentada [22] que usa duas possíveis heurísticas para guiar a busca por uma solução e uma abordagem neurogenética, solução híbrida associando características de redes neurais aumentadas e de algoritmos genéticos [23]; os autores mostraram que a intercalação de iterações do algoritmo genético e da rede neural pode levar a resultados melhores do que os obtidos pelas técnicas aplicadas separadamente.

2.3 Ambiente de Testes

As metaheurísticas estudadas e implementadas ao longo do desenvolvimento deste trabalho foram executadas com as mesmas instâncias de testes usadas por diversos trabalhos de referência para efeitos comparativos. Todos os programas foram codificados utilizando a linguagem C++ e, dependendo da versão paralela, também foram usa-

das as bibliotecas CUDA e MPI. Os testes sequenciais e utilizando GPGPU foram executados em máquinas com as seguintes configurações:

- Configuração 1:
  - Processador Intel I7-3770 de 3,40 GHz;
  - 24 GB de RAM;
  - Placa de vídeo NVIDIA GeForce GT 640 com 1 GB de memória e 384 núcleos de processamento;
  - Sistema operacional Ubuntu 12.10;
  - CUDA *toolkit* versão 7.0.
- Configuração 2:
  - Processador Intel I7-3770 de 3,40 GHz;
  - 8 GB de RAM;
  - Placa de vídeo NVIDIA GeForce GTX 680 com 2 GB de memória e 1536 núcleos de processamento;
  - Sistema operacional Ubuntu 12.04;
  - CUDA *toolkit* versão 4.2.
- Configuração 3:
  - Processador Intel I7-2600K de 3,40 GHz;
  - 8 GB de RAM;
  - Placa de vídeo NVIDIA GeForce GTX 580 com 1536 MB de memória e 512 núcleos de processamento;
  - Sistema operacional Ubuntu 12.04;
  - CUDA *toolkit* versão 4.2.

O *cluster beowulf* em que os programas em MPI foram executados utiliza um gerenciador de submissão de processos, o que torna transparente o modo como foi feita a alocação dos processadores para os processos. As configurações do *cluster* são as seguintes:

- 64 nós de processamento com 4 x 2,2 GHz Opteron Cores com 8 GB de RAM por nó;
- 4 nós com 12 x 2,53 GHz Intel Xeon E5649 com 24 GB de RAM por nó;
- 3 TB de armazenamento em disco;
- Interconexão usando gigabit ethernet.

Os programas desenvolvidos foram executados utilizando como entrada três conjuntos de testes: o conjunto SAC-94, a biblioteca ORLIB e o conjunto GK. O conjunto SAC-94 é baseado em problemas reais, provenientes de diferentes artigos, contendo instâncias com número de itens entre 10 e 105 e número de recursos entre 2 e 30. A biblioteca ORLIB, proposta por Chu e Beasley [8], é composta de instâncias de problemas com 5, 10 ou 30 recursos e 100, 250 ou 500 itens e, para cada combinação de recursos/itens considera-se, ainda, um "fator de *aperto*" α (*tightness factor*). O fator α determina como os valores do vetor de recursos disponíveis B se relacionam com a demanda por tais recursos e pode assumir os valor 0,25, 0,50 ou 0,75. Assim, os valores em B e em em R se correlacionam obedecendo à Equação (2.3):

$$b_j = \alpha * \sum_{i=1}^{n} r_{ji}, \text{para } j = 1, 2, ..., m \tag{2.3}$$

De acordo com essa equação, pode-se afirmar que a instância do problema se torna mais restrita quando α tende a zero. Para cada uma das configurações a biblioteca possui 10 instâncias de teste, totalizando 270 diferentes problemas. O conjunto GK, por sua vez, foi proposto mais recentemente por Glover e Kochenberger [32] e é composto por 11 instâncias maiores, cujos números de itens variam entre 100 e 2500 e os números de recursos variam entre 15 e 100.

Para medir a qualidade das soluções, utilizou-se o conceito de *gap*, que representa a porcentagem da diferença entre a solução obtida e a solução de referência, e é calculada com o uso da Equação (2.4):

$$gap = 100 * (valorReferencia - valorObtido)/valorReferencia \tag{2.4}$$

As soluções de referência são os melhores valores conhecidos para as instâncias de SAC-94 e de OR até o momento da escrita deste trabalho, muitos dos quais representando soluções provadamente ótimas; no caso das instâncias de GK, foram utilizados os valores obtidos pelo CPLEX, como apresentados em [72].

Devido à utilização de diferentes plataformas para execução, as comparações dos tempos dos programas implementados e dos trabalhos de referência foram realizadas utilizando um fator de correção calculado usando as frequências de *clock* dos processadores envolvidos. Tal fator é obtido dividindo as frequências de *clock* dos processadores descritos nos trabalhos de referência pela frequência da CPU dos testes deste trabalho; os valores calculados são, então, multiplicados pelos tempos de execução apresentados nos respectivos trabalhos.

2.4 Validação

As soluções obtidas pela implementações propostas foram validadas usando o teste de somas de *ranks* de Wilcoxon [76]. Neste teste, consideram-se duas amostras independentes provenientes de diferentes populações e se estabele a seguinte hipótese nula [76]:

H_0: as observações são provenientes da mesma população.

Se tal hipótese não puder ser rejeitada, considerando um determinado nível de significância, pode-se inferir que ambas as populações possuem a mesma mediana.

Para cada programa implementado, o teste foi aplicado considerando duas populações distintas: a primeira composta pelas médias dos resultados obtidos com as execuções e a segunda com os resultados de referência. O conjunto SAC-94 é composto por 54 instâncias, suficientemente grande para, de acordo com [76], considerar uma distribuição normal para o cálculo da estatística W; dessa forma, para cada implementação, calculou-se o p-valor equivalente aos resultados e aos valores de referência e, então, comparou-se tal valor com o nível de significância α considerado, neste caso, $\alpha = 0{,}05$. Caso o p-valor obtido for maior que o nível de significância, significa que a hipótese não pode ser rejeitada e, assim, os resultados de nossos algoritmos não são significativamente diferentes dos ótimos. O mesmo raciocínio foi aplicado às instâncias de teste da biblioteca OR, pois elas foram agrupadas de acordo com o número de recursos e o número de itens, resultando em 9 configurações compostas por 30 testes. No caso do conjunto de testes GK, a validação não pôde ser feita considerando uma distribuição normal, visto que o conjunto é composto por apenas 11 instâncias; assim, utilizou-se a menor soma W dos *ranks* de cada população (as soluções do algoritmo avaliado e as obtidas pelo CPLEX), tal valor é comparado a um valor crítico, obtido da tabela de soma de *ranks* de Wilcoxon (mostrada no Anexo B) considerando o nível de significância $\alpha = 0{,}05$, duas caudas e os tamanhos das duas populações. Como as duas populações têm tamanho 11, o valor crítico W_{crit} considerado nas validações realizadas é igual a 96. Se $W > W_{crit}$, a hipótese nula não pode ser rejeitada.

CAPÍTULO 3

Algoritmos Genéticos

3.1 Fundamentação Teórica

Um algoritmo genético (AG) é uma técnica de busca e otimização baseada em princípios da genética e da seleção natural [34]. A ideia principal dessa técnica consiste na existência de uma população composta por diversos indivíduos que evoluem de acordo com regras de seleção específicas até atingirem um estado que maximize sua adequação ao ambiente ou, ainda, até que um determinado número de gerações seja alcançado. A técnica foi desenvolvida ao longo das décadas de 1960 e 1970 por John Holland e descrita em seu livro de 1975 [37]. Algumas das principais vantagens apresentadas pelos AGs com relação a outras técnicas de otimização são [34]:

- Realizam otimizações com variáveis discretas ou contínuas;
- São capazes de lidar com um grande número de variáveis;
- Os resultados são apresentados como uma população de soluções e não como uma solução única;
- São facilmente adaptáveis a ambientes paralelos.

Um algoritmo genético, cujas etapas são ilustradas no Algoritmo 3.1 [17], pode ser visto como um método que combina uma seleção aleatória com a teoria da sobrevivência do mais forte, ou seja, os indivíduos mais fortes de uma população têm chances maiores de transferir seus genes para as próximas gerações. Para calcular a adaptabilidade de cada solução comparada às demais, utiliza-se uma função de *fitness* e, com base na avaliação dessa função para cada indivíduo, selecionam-se os "pais" dos indivíduos da próxima geração; esses passos são repetidos por um de-

terminado número de gerações ou até que uma determinada condição seja atingida.

Algoritmo 3.1: AlgoritmoGenéticoSimples()

```
1 Geração da população inicial;
2 Cálculo do fitness dos indivíduos;
3 repita
4 |   Seleção dos pais na população atual;
5 |   Reprodução;
6 |   Avaliação do fitness de cada filho;
7 |   Substituição de alguns indivíduos ou de toda a população pelos novos
  |   indivíduos;
8 até uma solução satisfatória ser encontrada;
```

Um algoritmo genético possui, minimamente, os seguintes elementos [52]: população de cromossomos, seleção de acordo com o *fitness*, reprodução para geração da nova população e mutação dos indivíduos da nova geração. Os cromossomos são mais comumente representados de forma binária, na qual cada gene pode assumir o valor 0 ou 1. A seleção é responsável por escolher, de acordo com a função de *fitness*, os cromossomos de uma população que serão utilizados para dar origem à próxima geração. A reprodução, conhecida também como *crossover*, é a operação responsável por realizar o cruzamento dos cromossomos escolhidos na etapa de seleção e, a partir deles, produzir novos indivíduos. A mutação consiste na alteração aleatória de alguns bits do cromossomo com uma determinada probabilidade, usualmente muito pequena [52].

As etapas de um algoritmo genético podem ser bastante dispendiosas quando o tamanho da população é grande e quando o número de gerações é elevado. Dessa forma, é relevante encontrar alternativas para executar algoritmos genéticos de maneira mais eficiente como, por exemplo, utilizando a programação paralela. Os algoritmos genéticos podem ser paralelizados considerando o modelo de programação paralela que se pretende adotar, destacando-se três alternativas [14]: população única global utilizando o paradigma mestre-discípulo, população única com granularidade fina e múltiplas populações com granularidade grossa, além de um modelo que pode misturar as alternativas anteriores (modelo híbrido hierárquico).

Na primeira alternativa, também chamada de método de paralelização global, existe uma população única e as atividades de cálculo e avaliação do *fitness*, bem como a aplicação dos operadores genéticos, são realizadas em paralelo pelos processadores. De maneira similar à abordagem sequencial, cada indivíduo é comparado com todos os demais da população e pode ser utilizado em conjunto com qualquer outro para gerar novos indivíduos. Um esquema do funcionamento deste tipo de paralelização é mostrado na Figura 3.1 [14].

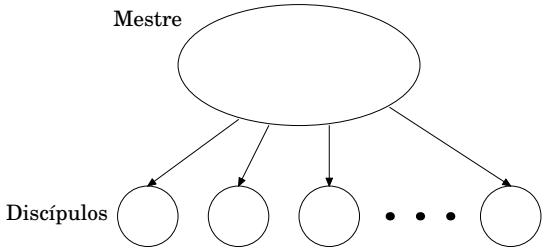


Figura 3.1: Esquema de funcionamento da paralelização global.

A avaliação do *fitness* de cada indivíduo é a etapa mais comumente executada em paralelo, uma vez que o cálculo do *fitness* de um indivíduo independe dos demais, o que faz com que a comunicação somente seja necessária para enviar os indivíduos para os processadores e para receber os valores calculados por cada processador.

O modelo de paralelização global independe da arquitetura sobre a qual é implementado, podendo ser utilizado tanto em ambientes com memória compartilhada quanto com memória distribuída. Com a utilização de memória compartilhada, cada processador pode ler os dados dos indivíduos pelo qual é responsável diretamente, enquanto que com memória distribuída, é necessário que o processador mestre explicitamente envie os indivíduos e colete os resultados calculados por cada processador. O número de indivíduos atribuídos a cada processador pode ser constante, entretanto, em alguns casos, pode ser necessário utilizar algum método de balanceamento dinâmico de carga.

Algoritmos genéticos de granularidade fina trabalham com uma única população; o que define como serão feitas as interações entre os indivíduos é a distribuição espacial dos elementos de processamento. Um indivíduo pode ser comparado e associado somente a um de seus adjacentes para realizar a seleção e a reprodução; entretanto, como os elementos de processamento formam uma grade conexa de interligação, as boas soluções irão se propagar por toda a população. É bastante comum organizar os elementos de processamento em grades bidimensionais. Um exemplo de distribuição dos elementos de processamento é ilustrado na Figura 3.2 [14].

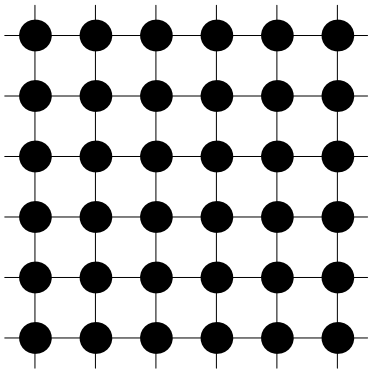


Figura 3.2: Uma distribuição de processadores no modelo de granularidade fina.

A paralelização com granularidade grossa, também conhecida como *paralelização com múltiplos demes*, é caracterizada pela existência de um conjunto de subpopulações relativamente grandes distribuídas entre os elementos de processamento. Uma possível configuração do modelo de granularidade grossa pode ser vista na Figura 3.3 [14].

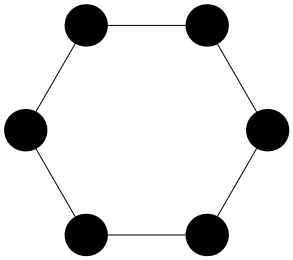


Figura 3.3: Distribuição de processadores no modelo de granularidade grossa.

Nesse tipo de paralelização, é essencial que exista uma maneira de evitar que as subpopulações evoluam de maneira totalmente isoladas das demais, como se fossem apenas diversas instâncias do algoritmo genético sequencial. Para evitar esse comportamento, utiliza-se o mecanismo de *migração*, que consiste em trocar indivíduos entre as subpopulações após uma determinada quantidade de gerações. Surgem novos parâmetros que devem ser definidos antes da execução de um algoritmo genético com granularidade grossa tais como: tamanho das subpopulações, frequência das migrações, escolha dos elementos a serem migrados e quais devem ser substituídos com a chegada dos novos elementos, entre outros.

O processo de migração pode ser realizado de diferentes maneiras, sendo o *modelo de ilhas* (Figura 3.4) [46] um dos mais utilizados. Neste modelo, quando é chegado o momento da migração, cópias dos melhores elementos de cada processador – aqui chamado de *ilha* – são enviados para outro processador que, em troca, envia também cópias de seus melhores elementos; cada processador utiliza os elementos recebidos para substituir os elementos com menor aptidão de sua população local.

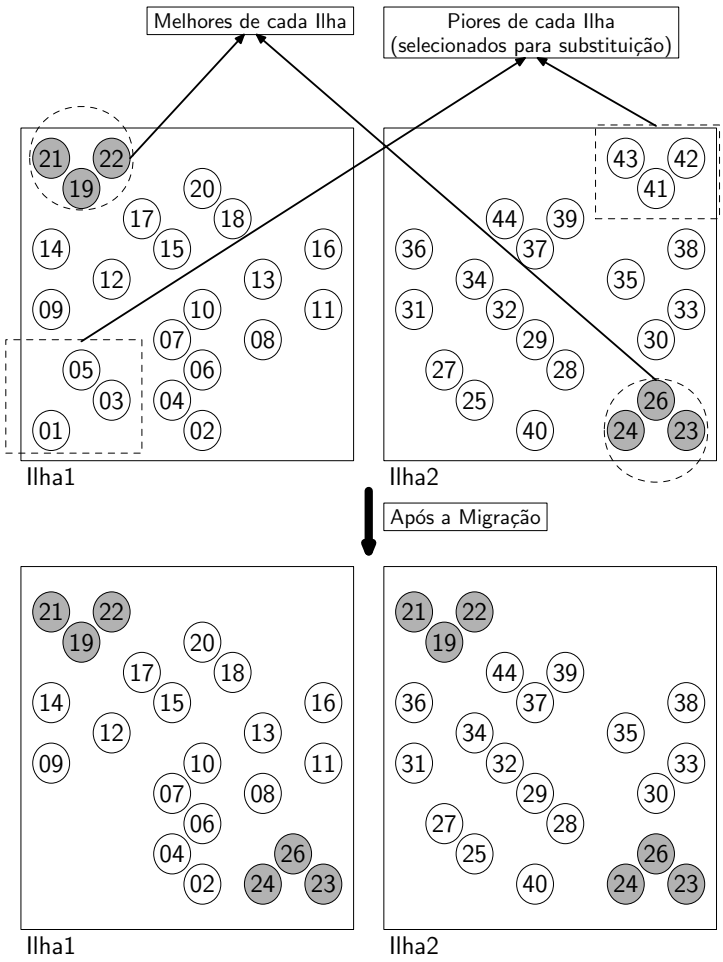


Figura 3.4: Modelo de ilhas de migração.

3.2 Implementações

Foram implementadas uma versão sequencial do algoritmo e duas paralelas, uma utilizando múltiplos processadores distribuídos que se comunicam por troca de mensagens e outra unidades de processamento gráfico (GPUs). Em todas as implementações, o tamanho da população e o número de gerações são fornecidos como entrada, bem como o intervalo de migração nas implementações paralelas. A representação do problema é feita através de uma população contendo indivíduos representados por cromossomos com genes binários. Cada cromossomo, ou indivíduo, contém n genes, onde n é o número de itens disponíveis, e representa uma possível solução para o problema.

A população inicial, gerada aleatoriamente, e as subsequentes são compostas apenas por indivíduos que representam uma solução viável para o problema, ou seja, uma solução válida. Para garantir a viabilidade da solução, logo após sua geração, verifica-se se ela satisfaz as restrições do problema e, caso não satisfaça, uma operação de correção é realizada na qual um item aleatório é "abandonado"; esse processo se repete até que a solução se torne viável.

A seleção é feita pela escolha dos dois melhores indivíduos da população atual, os quais são utilizados para dar origem a todos os indivíduos da nova geração. A estratégia de reprodução utilizada é o *crossover* de um ponto, na qual um *ponto de corte* é escolhido aleatoriamente e os dois filhos são gerados com base em tal ponto, como pode ser visto na Figura 3.5. Um dos filhos possui seus primeiros genes copiados do primeiro pai até o ponto de corte e os demais genes são copiados do segundo pai a partir da posição imediatamente posterior ao ponto de corte; o outro filho é formado da maneira inversa, os primeiros genes vêm do segundo pai e os demais do primeiro. Para a mutação utilizou-se a estratégia invertida, na qual um gene é selecionado aleatoriamente no cromossomo e seu valor é invertido.

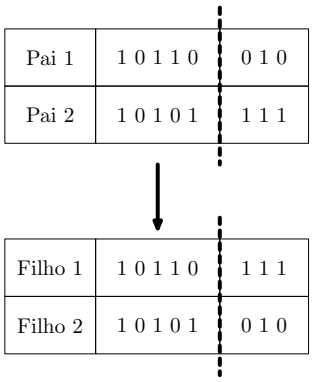


Figura 3.5: *Crossover* de um ponto.

A implementação paralela para múltiplos processadores foi realizada utilizando a biblioteca de troca de mensagens MPI (*Message Passing Interface*) [56]. A abordagem implementada foi a de granularidade grossa com múltiplas subpopulações, na qual cada processador executa o algoritmo sequencial. O processo de migração obedece a uma topologia de anel e, a cada intervalo de migração, os processadores trocam indivíduos com seus adjacentes – o sucessor e o antecessor. O elemento que cada processador envia para seu sucessor é aquele que possui o melhor *fitness* em sua população local; o

elemento recebido de seu antecessor é utilizado para substituir o indivíduo local com menor *fitness*.

Os parâmetros de entrada do algoritmo são: o arquivo contendo os dados da instância (arq), o número de processadores (p), o tamanho da população ($tamPop$), o número de gerações ($numGer$) e o intervalo de migração ($intMigr$). O Algoritmo 3.2 ilustra os passos seguidos nesta implementação.

Algoritmo 3.2: AG_MPI(arq, p, $tamPop$, $numGer$, $intMigr$)

```
1  Processador 0 lê arq e os distribui entre os demais processadores;
2  Cada processador gera tamPop/p indivíduos da população inicial local;
3  Cada processador calcula o fitness dos indivíduos;
4  repita
5  |  Seleciona pais na população atual;
6  |  Realiza a reprodução e possível mutação;
7  |  Avalia o fitness da cada filho;
8  |  Substitui toda a população pelos novos indivíduos;
9  |  se iteração é múltiplo de intMigr então
10 |  |  Cada processador envia seu melhor indivíduo para o seu sucessor e recebe
   |  |  o melhor indivíduo de seu antecessor;
11 |  |  Cada processador substitui indivíduo com menor fitness pelo recebido;
12 |  fim
13 até cada processador atingir numGer gerações;
14 Processador 0 coleta o melhor indivíduo de cada processador e o resultado é o
   melhor de todos os indivíduos;
```

Na abordagem usando GPGPU, dividiu-se a população em ilhas, representadas por blocos de *threads* nos quais cada *thread* representa um indivíduo. Na etapa de geração da população inicial, cada *thread* é responsável por gerar e calcular o *fitness* de um único indivíduo. O processo de seleção é realizado por uma única *thread* de cada bloco e, então, metade das *threads* criadas realiza o *crossover* dos pais, gerando dois indivíduos e, possivelmente, aplica o operador de mutação aos indivíduos gerados e calcula o seu *fitness*. A cada geração, verifica-se se o intervalo de migração foi alcançado e, neste caso, realiza-se a migração em anel. O Algoritmo 3.3 ilustra os passos do programa desenvolvido.

Algoritmo 3.3: AG_GPGPU(*arq*, *tamPop*, *numGer*, *intMigr*)

```
1  CPU lê arq;
2  Criação de n threads na GPGPU para gerar um indivíduo da população inicial;
3  Cada thread calcula o fitness do indivíduo por ela gerado;
4  repita
5      Uma thread de cada bloco seleciona os pais na população atual;
6      Metade das threads realiza a reprodução, mutação e avaliação do fitness de
       dois novos indivíduos;
7      Toda a população é substituída pelos novos indivíduos;
8      se iteração é múltiplo de intMigr então
9          Cada ilha envia seu melhor indivíduo para o seu sucessor e recebe o
           melhor indivíduo de seu antecessor;
10         Cada ilha substitui indivíduo com menor fitness pelo recebido;
11     fim
12 até o número de gerações numGer ser alcançado por todas as threads;
13 População final é copiada para CPU, resultado é o melhor indivíduo;
```

3.3 Resultados

Uma execução de um algoritmo genético é caracterizada por um conjunto de fatores, tais como tamanho da população, número de gerações e os métodos usados para geração da população inicial, seleção, *crossover*, mutação, entre outros. No caso das implementações propostas, apenas os dois primeiros parâmetros podem ser configurados em tempo de execução e diferentes configurações foram testadas. Para avaliar a influência do tamanho da população e do número de gerações na qualidade dos resultados, foram testadas populações de tamanho 100, 500, 1000 e 5000 e números de gerações iguais a 50, 100, 500, 1000, 5000 e 10000. Para cada combinação de tamanho população e número de gerações, foram medidos os *gaps* obtidos e os tempos de execução e, após avaliação dos resultados obtidos, notou-se que o aumento no número de execuções não causou melhora na qualidade das soluções obtidas que compensassem o aumento no tempo de execução acarretado pelas gerações adicionais. Com relação à variação no tamanho da população, notou-se que o aumento no número de indivíduos levou a uma melhora mais evidente do que a ampliação no número de gerações, ainda que suave, na qualidade das soluções alcançadas, embora ainda não na mesma proporção que o aumento no tempo de execução da heurística.

Partindo dessas observações, foram utilizadas 500 gerações para as execuções de ambas as implementações e uma população composta por 4096 indivíduos. Optou-se por esse tamanho de população por ser o valor mais próximo de 5000 (que levou aos melhores resultados nas avaliações realizadas) e que também é uma potência de 2. A ideia de usar uma potência de 2 se deve ao fato de as GPU organizarem suas *threads* dessa maneira. Os resultados obtidos pela execução do programa sequencial, usando o ambiente de testes com a configuração 2, com os três conjuntos de teste são apresentados nas Tabelas 3.1, 3.2 e 3.3. As tabelas mostram, para cada configuração do conjunto em questão, o número de vezes que a solução de referência foi alcançada (coluna Ótimos), o menor *gap* alcançado nas execuções realizadas (coluna *Gap* mínimo), a média dos *gaps* obtidos em todas as execuções (coluna *Gap* médio), a média dos

desvios padrões dos *gaps* alcançados para cada configuração (DP) e o tempo de execução médio de todas as execuções (coluna Tempo).

Tabela 3.1: Resultados da implementação sequencial do algoritmo genético para as instâncias de SAC-94.

		AG Sequencial				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	29	0,0000	0,7036	0,7936	2,9069
PB	6	96	0,0000	0,5360	0,5813	8,5660
PET	6	129	0,0000	0,1388	0,1166	3,3961
SENTO	2	22	0,0000	0,1071	0,0958	29,1768
WEING	8	204	0,0000	0,0647	0,1289	3,4049
WEISH	30	735	0,0000	0,0625	0,1029	7,7922

Tabela 3.2: Resultados da implementação sequencial do algoritmo genético para as instâncias de ORLIB agrupadas por configurações.

			AG Sequencial				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	0	0,0496	1,5093	0,5828	11,3661
		0,50	0	0,0690	0,9075	0,3309	10,2153
		0,75	0	0,0196	0,6370	0,2506	9,2226
	250	0,25	0	0,7463	2,2974	0,6129	28,3209
		0,50	0	0,4290	1,1432	0,2951	25,2728
		0,75	0	0,1390	0,6821	0,1927	22,9817
	500	0,25	0	1,3317	2,3659	0,4559	57,3499
		0,50	0	0,5318	1,1349	0,2435	50,7038
		0,75	0	0,3364	0,6608	0,1416	45,3508
10	100	0,25	0	0,0911	2,2514	0,7484	18,0457
		0,50	0	0,3167	1,2875	0,4756	16,6985
		0,75	2	0,0000	0,7967	0,3171	14,8830
	250	0,25	0	1,3926	2,9376	0,6431	44,9138
		0,50	0	0,6816	1,4895	0,4137	40,7100
		0,75	0	0,2955	0,7796	0,2369	36,5366
	500	0,25	0	1,4794	2,8409	0,6057	90,4710
		0,50	0	0,7140	1,4134	0,3379	81,9774
		0,75	0	0,3295	0,7228	0,1651	73,2840
30	100	0,25	1	0,0000	2,3350	0,7554	42,7396
		0,50	0	0,0632	1,3153	0,5099	42,0789
		0,75	1	0,0000	0,7713	0,2712	37,2839
	250	0,25	0	1,2573	2,8878	0,7392	111,0889
		0,50	0	0,5702	1,4669	0,4826	102,7924
		0,75	0	0,3069	0,7826	0,2648	92,0179
	500	0,25	0	1,5626	2,7411	0,7080	221,7397
		0,50	0	0,7063	1,3617	0,4173	208,5596
		0,75	0	0,3280	0,7078	0,2068	183,8023

Tabela 3.3: Resultados da implementação sequencial do algoritmo genético para as instâncias de GK.

		AG Sequencial				
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
15	100	0	0,6638	1,2843	0,3169	19,9953
25	100	0	0,6822	1,2986	0,2832	30,1836
25	150	0	0,8133	1,4869	0,3542	45,5740
50	150	0	0,9017	1,4866	0,2487	85,1817
25	200	0	0,9528	1,4785	0,2631	60,5840
50	200	0	1,0428	1,4864	0,2568	113,0850
25	500	0	1,1397	1,7139	0,2712	150,9935
50	500	0	1,4304	1,7312	0,1747	282,6370
25	1500	0	1,7371	2,0275	0,1450	459,5097
50	1500	0	1,7419	1,9122	0,0820	850,4931
100	2500	0	1,4943	1,6303	0,0561	2751,2437

Para execução da versão paralela, foram testadas diferentes configurações de *grid* para totalizar 4096 *threads* e a que apresentou os melhores *speedups* e qualidades na média é composta por 32 blocos (ou ilhas) de 128 *threads*. Os resultados obtidos e os *speedups* com relação à versão sequencial, usando o mesmo número de gerações da execução sequencial e a configuração 2 do ambiente de testes, são mostrados nas Tabelas 3.4, 3.5 e 3.6.

Tabela 3.4: Resultados da implementação usando GPGPGU do algoritmo genético para as instâncias de SAC-94.

		AG GPGPU					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	27	0,0000	0,5185	0,5477	0,3873	7,5490
PB	6	116	0,0000	0,2740	0,3683	1,0784	8,3890
PET	6	127	0,0000	0,0777	0,0727	0,4397	9,1231
SENTO	2	16	0,0000	0,1162	0,1137	3,8998	7,4797
WEING	8	185	0,0000	0,0508	0,0867	0,4647	8,0183
WEISH	30	740	0,0000	0,0346	0,0660	0,9408	8,5474

Tabela 3.5: Resultados da implementação usando GPGPGU do algoritmo genético para as instâncias de ORLIB agrupadas por configurações.

			AG GPGPU					
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	0	0,1401	1,9113	0,6603	1,1658	9,7531
		0,50	1	0,0000	1,0488	0,3712	1,0377	9,8485
		0,75	0	0,0167	0,7105	0,2522	0,9062	10,1821
	250	0,25	0	1,4417	3,1033	0,6895	2,9630	9,5594
		0,50	0	0,9641	1,7541	0,3895	2,5856	9,7765
		0,75	0	0,5062	1,1206	0,2878	2,2460	10,2412
	500	0,25	0	2,1399	4,1437	0,7623	5,9788	9,5940
		0,50	0	1,4121	2,5294	0,4612	5,0128	10,1154
		0,75	0	0,9273	1,6374	0,3121	4,3589	10,4079
10	100	0,25	0	0,0390	2,5958	0,8142	1,7761	10,1737
		0,50	0	0,2234	1,4155	0,4686	1,6961	9,8487
		0,75	0	0,0510	0,8617	0,3198	1,4701	10,1325
	250	0,25	0	1,7462	3,5687	0,7325	4,6590	9,6429
		0,50	0	1,0198	2,0320	0,4562	4,1680	9,7686
		0,75	0	0,4700	1,1775	0,3064	3,5996	10,1524
	500	0,25	0	2,8263	4,6606	0,7167	9,1447	9,8946
		0,50	0	1,7174	2,8740	0,5525	8,1356	10,0777
		0,75	0	0,9864	1,7879	0,3636	7,0768	10,3575
30	100	0,25	0	0,3622	2,5449	0,6391	4,1825	10,2286
		0,50	0	0,2715	1,4063	0,4426	4,1401	10,1557
		0,75	4	0,0000	0,8116	0,2911	3,6743	10,1515
	250	0,25	0	1,8775	3,4741	0,7626	11,2257	9,8980
		0,50	0	0,7834	1,9618	0,5323	10,3490	9,9415
		0,75	0	0,5175	1,1861	0,2964	9,2310	9,9747
	500	0,25	0	2,4900	4,4472	0,7808	22,3857	9,9067
		0,50	0	1,7589	2,8934	0,5012	20,6742	10,0917
		0,75	0	1,0993	1,7460	0,3085	17,7672	10,3463

Tabela 3.6: Resultados da implementação usando GPGPGU do algoritmo genético para as instâncias de GK.

		AG GPGPU					
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
15	100	0	0,6638	1,1648	0,3038	1,6231	12,3192
25	100	0	0,5558	1,2094	0,2693	2,4269	12,4371
25	150	0	0,8487	1,5252	0,3019	3,8437	11,8568
50	150	0	1,3005	1,6814	0,1774	7,2921	11,6814
25	200	0	1,3365	1,8014	0,2489	4,9803	12,1647
50	200	0	1,2774	1,6076	0,1566	9,7028	11,6549
25	500	0	1,7278	2,1627	0,2139	13,3071	11,3468
50	500	0	1,6963	1,9625	0,1609	27,9550	10,1104
25	1500	0	2,4671	2,7851	0,1220	44,5040	10,3251
50	1500	0	2,0369	2,2336	0,1015	94,8116	8,9703
100	2500	0	1,7169	1,8265	0,0592	389,3016	7,0671

Como atestam os valores apresentados nas tabelas, a versão GPGPU apresentou *speedups* de 9,3802 em média e, embora tenha havido redução da qualidade com relação à versão sequencial, isso comprovou que o uso de paralelismo possibilita a obtenção de bons resultados em tempo reduzido.

Como dito anteriormente, os resultados atingidos pela implementação MPI não se mostraram interessantes, isso é explicado pelo fato de que as qualidades obtidas não suplantaram as demais implementações e os tempos de execução não foram suficientemente reduzidos. Na busca por melhores tempos de execução, foram realizadas testes usando 2, 8, 32 e 64 processadores e, à medida que o tempo de execução diminuía, a qualidade dos resultados também era prejudicada. Percebeu-se, que para alcançar os mesmos tempos obtidos em CUDA, era necessário sacrificar a qualidade e, a partir desse momento, os experimentos com MPI foram encerrados. Os gráficos das Figuras 3.6 e 3.7 mostram, respectivamente, as variações nos *gaps* e nos tempos de execução do AG usando MPI com diferentes números de processadores para as instâncias de ORLIB agrupadas por números de itens.

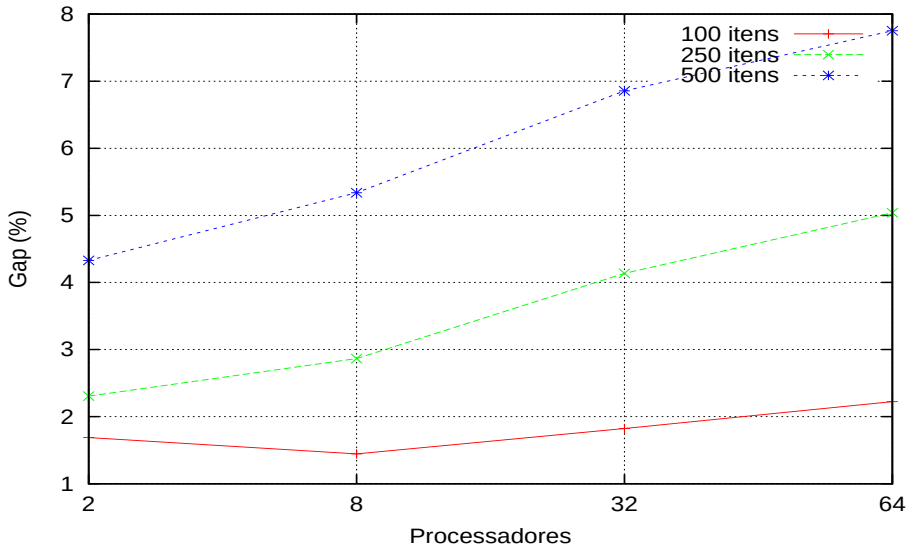


Figura 3.6: *Gaps* do AG usando MPI agrupados por número de itens.

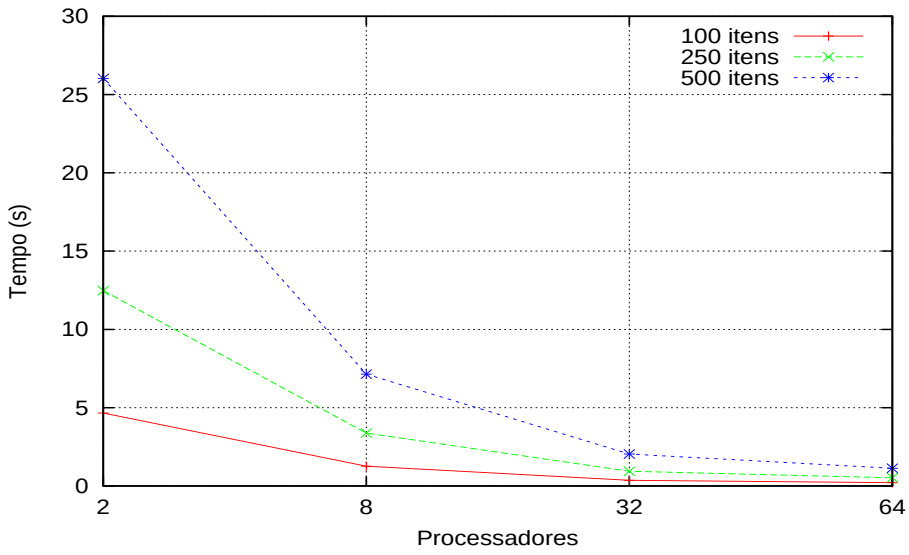


Figura 3.7: Tempos do AG usando MPI agrupados por número de itens.

Para validar os resultados obtidos, aplicou-se o teste de Wilcoxon de soma de *ranks*. Os p-valores computados para instâncias de SAC-94 foram 0,4584 e 0,4475, respectivamente para as versões sequencial e paralela; os valor de W, para as instâncias de GK, foi 118 para ambas as versões. A Tabela 3.7 mostra os p-valores calculados para as configurações das instâncias de OR.

Tabela 3.7: p-valores calculados para os resultados do algoritmo genético para cada configuração de testes de ORLIB.

		p-valor	
m	n	Sequencial	GPGPU
5	100	0,2529	0,2344
	250	0,1155	0,0779
	500	0,0869	**0,0282**
10	100	0,2039	0,1836
	250	0,0966	0,0621
	500	0,0658	**0,0192**
30	100	0,1610	0,1574
	250	0,0639	**0,0489**
	500	**0,0418**	**0,0179**

Pelos p-valores calculados, os resultados do algoritmo genético não são, em sua maioria, significativamente diferentes das melhores soluções conhecidas, exceto nas configurações cujos p-valores estão em negrito na tabela.

Embora os resultados das implementações sequencial e usando GPGPU tenham levado a resultados melhores do que os obtidos pela implementação usando MPI, elas não obtiveram resultados satisfatórios quando comparadas às demais heurísticas estudadas ao longo deste trabalho. Por esse motivo, diferentes alternativas para as etapas constituintes do AG merecem ser estudadas como forma de melhorar o seu desempenho. Um primeiro passo nesse sentido foi implementar uma segunda versão do AG com praticamente as mesmas configurações utilizadas por Chu e Beasley [17], cujas estratégias de seleção, reprodução, mutação e de substituição da população a cada geração diferem das que foram inicialmente implementadas neste trabalho.

Na nova versão a seleção dos pais é feita usando o método de torneio binário, no qual duas coleções são criadas, cada uma composta por dois indivíduos escolhidos aleatoriamente da população, e, então, o melhor indivíduo de cada coleção é escolhido. Na reprodução utiliza-se o operador de *crossover* uniforme. A mutação, realizada em uma taxa de 1 ou 2 bits por indivíduo, realiza a inversão de bits selecionados aleatoriamente. O operador de reparação utilizado difere da versão de Chu e Beasley e se baseia na ideia de pseudoutilidades, enquanto, no caso de Chu e Beasley, foi adotado o conceito de dualidade substituta proposta por Pirkul [57], a qual utiliza os valores das variáveis duais provenientes da relaxação PL da instância do problema obtida pelo CPLEX, por exemplo. Na implementação realizada para efeitos comparativos, os valores de pseudoutilidades utilizados foram os mesmos das implementações de redes neurais aumentadas, GRASP e da abordagem híbrida de ambas (a pseudoutilidade de cada item é calculada de acordo com a Equação 4.6, que será descrita detalhadamente no Capítulo 4).

Com base na pseudoutilidade, o indivíduo gerado pelo *crossover*, e eventual mutação, pode ser "reparado", caso não represente uma solução viável. Nesse caso, obtém-se uma ordenação dos itens de acordo com sua pseudoutilidade e, baseado nessa ordem, realiza-se um processo de duas fases: na primeira, itens são removidos em ordem não-decrescente de pseudoutilidade até a solução se tornar viável, na segunda, itens são adicionados em ordem não-crescente de pseudoutilidade na tentativa de melhorar a qualidade da solução, até que não se possa mais adicionar outro item.

Uma diferença fundamental nessa nova abordagem implementada está no fato de que a cada geração, o processo de reprodução gera um único novo indivíduo e ele substitui o pior indivíduo da população. Os resultados obtidos para as instâncias de OR com 1.000.000 de gerações do AG e considerando uma população com 100 indivíduos, como realizado por Chu e Beasley, são apresentados na Tabela 3.8.

Tabela 3.8: Resultados da nova abordagem do algoritmo genético sequencial para as instâncias de ORLIB agrupadas por configurações.

			AG baseado em Chu e Beasley [17]				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	292	0,0000	0,0032	0,0078	7,2946
		0,50	264	0,0000	0,0036	0,0042	7,3886
		0,75	178	0,0000	0,0184	0,0141	6,9687
	250	0,25	25	0,0000	0,0906	0,0371	14,7180
		0,50	0	0,0018	0,0597	0,0221	15,7574
		0,75	1	0,0000	0,0358	0,0137	15,9504
	500	0,25	0	0,0168	0,1206	0,0318	28,2968
		0,50	1	0,0000	0,0646	0,0192	29,4568
		0,75	0	0,0044	0,0491	0,0147	29,9896
10	100	0,25	191	0,0000	0,0600	0,0667	7,9504
		0,50	133	0,0000	0,0650	0,0399	8,5416
		0,75	163	0,0000	0,0446	0,0304	8,7826
	250	0,25	4	0,0000	0,1834	0,0718	17,0356
		0,50	0	0,0036	0,1067	0,0330	18,3509
		0,75	6	0,0000	0,0817	0,0238	18,3035
	500	0,25	0	0,0808	0,2230	0,0458	32,6569
		0,50	0	0,0389	0,1496	0,0304	33,5655
		0,75	0	0,0131	0,0868	0,0205	34,4361
30	100	0,25	179	0,0000	0,1512	0,1397	10,9716
		0,50	123	0,0000	0,0670	0,0342	13,4647
		0,75	139	0,0000	0,0471	0,0244	14,0741
	250	0,25	18	0,0000	0,3411	0,1094	24,1053
		0,50	0	0,0317	0,1832	0,0555	28,7564
		0,75	1	0,0000	0,1347	0,0419	31,7766
	500	0,25	0	0,1803	0,4195	0,0742	44,7698
		0,50	0	0,0824	0,2101	0,0413	54,3804
		0,75	0	0,0795	0,1697	0,0296	60,1119

Os resultados da nova implementação levaram a resultados muito superiores aos obtidos na primeira versão, o que sugere que a abordagem elitista, e gulosa, presente na primeira versão, a qual escolhe os dois melhores indivíduos e, a partir deles, executa a troca de todos os indivíduos da população a cada geração, faz com que soluções potencialmente boas sejam abandonadas ao longo do processo. Essa característica torna a busca por soluções concentrada em uma única vizinhança em que os indivíduos possuem as mesmas características. Outro aspecto que diferencia as duas abordagens é que o processo de trocar todos os indivíduos a cada iteração faz com que o uso do mesmo número de gerações proposto por Chu e Beasley seja inviável na primeira versão

implementada, pois o tempo de execução necessário seria muito alto. Por outro lado, a opção por realizar apenas uma substituição por geração reduz as possibilidades que podem ser exploradas na versão paralela, fazendo com que não seja vantajoso utilizar a mesma estratégia de paralelização usada na primeira implementação, o que leva à necessidade do estudo de outra alternativa.

Para ressaltar o quão promissores os novos resultados se mostraram, o gráfico da Figura 3.8 ilustra um comparativo com os resultados obtidos por Chu e Beasley e mostrados em [17].

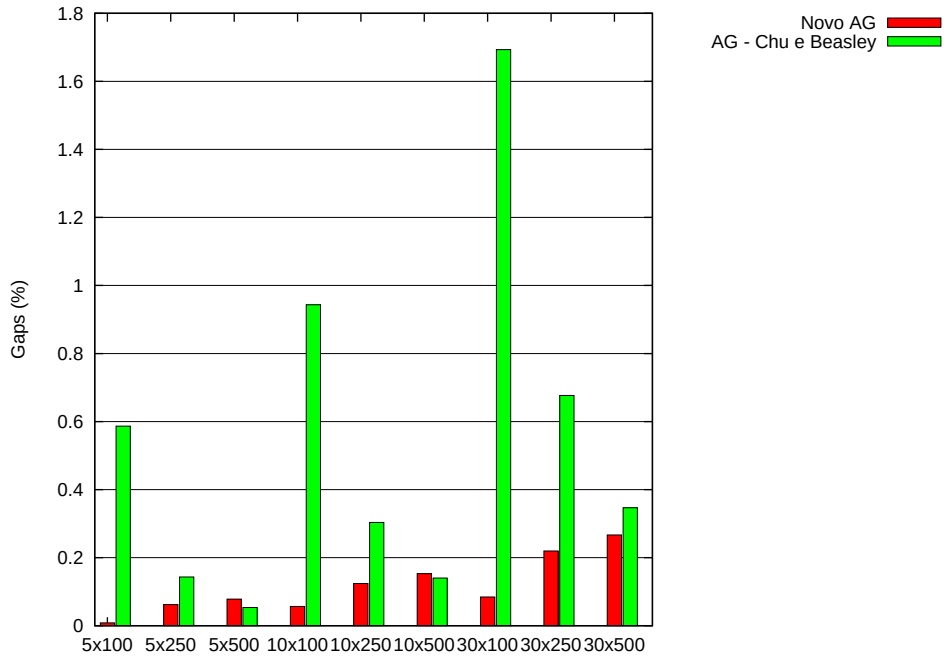


Figura 3.8: *Gaps* do novo AG para as instâncias de ORLIB comparados ao AG de Chu e Beasley.

Pela comparação dos resultados, a nova implementação proposta foi capaz de alcançar resultados melhores do que a versão original de Chu e Beasley em todas as configurações, exceto em 5×500 e 10×500, entretanto, mesmo nessas configurações a qualidade das soluções não foi muito inferior às obtidas por eles.

3.4 Contribuições

Como contribuições resultantes das implementações de algoritmos genéticos descritas neste capítulo, destacam-se:

- Publicação de artigo no XLVI Simpósio Brasileiro de Pesquisa Operacional (SBPO 2014) [19] com os resultados da primeira versão implementada;
- Proposta de nova alternativa para cálculo das pseudoutilidades, independente de outros *solvers*, na segunda versão;
- Excelente desempenho em instâncias menores, gradualmente reduzida com o aumento das instâncias;

- Bons *speedups* na versão paralela.

CAPÍTULO 4

Redes Neurais Aumentadas

4.1 Fundamentação Teórica

Inspiradas na hipótese de que o funcionamento cerebral consiste basicamente em atividades eletroquímicas em redes de células cerebrais especiais, denominadas *neurônios*, algumas das primeiras pesquisas da área de inteligência artificial foram focadas no desenvolvimento de *redes neurais artificiais*.

Uma rede neural artificial, ou simplesmente rede neural, é uma coleção de unidades, os denominados *neurônios artificiais*, com conexões direcionadas e cujo funcionamento é determinado pela sua topologia e pelas propriedades de seus neurônios [66]. Um modelo matemático desenvolvido em 1943 por McCulloch e Pitts [51] para representar um neurônio é mostrado na Figura 4.1 [66].

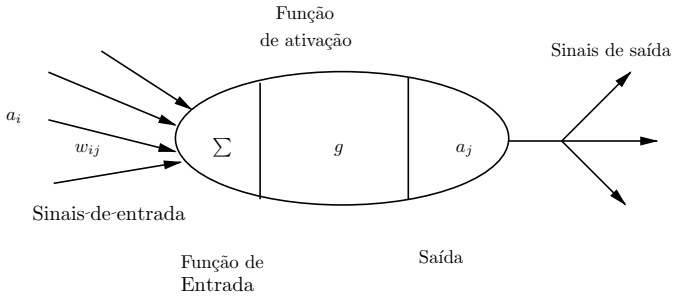


Figura 4.1: Modelo matemático para um neurônio.

Uma conexão do neurônio i para o neurônio j tem como funcionalidade propagar a *ativação* a_i de i para j. Além disso, cada conexão possui também um *peso* numérico w_{ij} associado, o qual determina a força e o sinal da conexão. Cada neurônio j computa a soma ponderada de seus sinais de entrada de acordo com a Equação (4.1) [66]:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i. \tag{4.1}$$

Para produzir a saída, o neurônio aplica uma *função de ativação g* ao resultado obtido, a qual depende da aplicação desenvolvida, como representado na Equação (4.2) [66]:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{ij} a_i). \tag{4.2}$$

Com relação à topologia, as redes neurais são compostas por uma ou mais camadas de neurônios através das quais os sinais de entrada/saída fluem, até que a saída da rede seja produzida. De acordo com o sentido do fluxo desses sinais, as redes neurais podem ser classificadas como *feed-forward* ou *recorrentes*. Uma rede *feed-forward* possui conexões em apenas uma direção, ou seja, sem realimentação. Uma rede recorrente, por sua vez, utiliza as saídas de seus neurônios para alimentar outros neurônios pertencentes à mesma camada ou a camadas anteriores.

Uma *rede neural aumentada* é uma metaheurística que consiste na associação de uma heurística ao funcionamento habitual de uma rede neural, essa heurística é utilizada para tomar decisões ao longo das camadas da rede [6]. Nessa abordagem, define-se a topologia da rede neural levando em consideração o funcionamento da heurística, a qual determina o comportamento das funções de entrada, saída e ativação de cada uma das camadas componentes da rede. A variabilidade nas buscas pelas soluções é implementada com alterações nos pesos entre os elementos de processamento.

Neste trabalho foi utilizada a abordagem de redes neurais aumentadas apresentada por Deane e Agarwal [22, 23] e previamente aplicada ao problema de escalonamento de tarefas [3, 4]. A ideia principal nessa abordagem é associar à estrutura da rede neural uma heurística para resolução do problema. No caso do problema da mochila multidimensional, pode-se utilizar, por exemplo, as propostas por Senju e Toyoda [68] e por Kochenberger, McCarl e Wyman [42].

Na primeira heurística, referenciada pela sigla ST, os autores propõem que a mochila inicialmente contenha todos os itens e, a partir dessa configuração, os elementos sejam retirados, um a um, até que a mochila se torne viável, ou seja, até que os itens restantes na mochila não ultrapassem as capacidades disponíveis dos recursos. A segunda heurística, referenciada pela sigla KMW, age de maneira inversa, a mochila é iniciada vazia e os itens são acrescentados a ela até que a tentativa de adicionar um item extrapole a capacidade de pelo menos um dos recursos. Em ambas as heurísticas, é necessário que se estabeleça a ordem pela qual os itens são retirados ou adicionados à mochila, para isso utiliza-se como métrica a *pseudoutilidade* dos itens, a qual é calculada para cada item considerando seu valor e o quanto ele usa de cada recurso.

A pseudoutilidade $psUt$ na heurística ST é calculada utilizando a Equação (4.3):

$$psUt_i = v_i / \{\sum_{j=1}^{m} cs_j * r_{ji}\}, i = 1, ..., n \tag{4.3}$$

onde o vetor CS representa, em porcentagem, o quanto a capacidade de cada recurso é extrapolada quando todos os itens estiverem na mochila. A cada etapa da busca pela solução, retira-se o item com a menor pseudoutilidade. A heurística KMW, por sua vez, usa a Equação (4.4) para computar a pseudoutilidade:

$$psUt_i = v_i / \{\sum_{j=1}^{m} r_{ji}/rc_j\}, i = 1, ..., n \tag{4.4}$$

onde RC é um vetor com m posições que armazena para cada recurso o quanto de sua capacidade ainda está disponível. A escolha do item a ser adicionado à mochila é feita em ordem não-crescente de pseudoutilidade; quando o item escolhido tornar a mochila inviável, considera-se que a solução foi encontrada. Em uma variante dessa heurística, busca-se entre os demais itens algum outro com menor pseudoutilidade que possa ser adicionado. A implementação do KMW descrita na Seção 4.2 utiliza essa variante.

A rede neural que utiliza uma das duas heurísticas executa um número máximo predefinido de *épocas*. Uma época, por sua vez, é uma sequência de *iterações* realizadas para obter uma solução viável, a cada iteração um item é retirado ou adicionado à mochila, conforme a heurística em uso. A rede neural retorna como resultado a melhor solução obtida ao longo de suas épocas.

Para fazer com que as soluções variem entre uma época e outra, utiliza-se um *vetor de pesos* que visa "perturbar" a ordem na qual os itens são considerados. O vetor de pesos $W = \{w_1, w_2, \ldots, w_n\}$, inicialmente com o valor 1 em todas as posições, é combinado com o vetor de custos V para realizar o cálculo das pseudoutilidades, ou seja, no local onde as heurísticas puras utilizam o valor v_i a rede neural aumentada utiliza $v_i * w_i$. O vetor de pesos para a k-ésima iteração é obtido utilizando a Equação (4.5):

$$w_i^k = w_i^{k-1} \pm \alpha * gap * \beta_i * v_i, i = 1, ..., n \tag{4.5}$$

onde α é a taxa de aprendizado da rede e β_i é um valor aleatório entre 0 e 1 gerado para cada item; além disso, se $0 < \beta_i \leq 0{,}5$, a atualização é realizada com a subtração, caso contrário utiliza-se a adição. O valor do *gap* é calculado de acordo com a Equação (2.4) mostrada no Capítulo 2.

A topologia da rede neural, ilustrada na Figura 4.2, é a mesma utilizada por [22, 23] e é composta por três camadas:

- Camada de entrada: um único nó que determina a atividade que a rede deve executar;
- Camada escondida: representa os itens e possui n nós, um para cada item do problema;
- Camada de saída: é a camada que determina qual item vai ser removido/adicionado à mochila e, consequentemente, desativado para as próximas iterações dentro da mesma época.

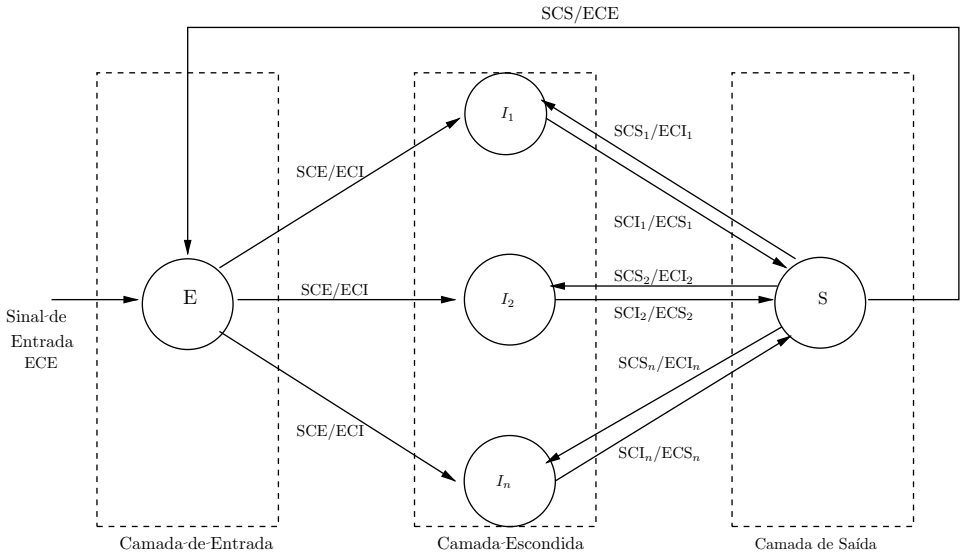


Figura 4.2: Topologia da rede neural.

Cada camada recebe sinais de entrada das camadas anteriores e envia sinais de saída para as camadas subsequentes, na figura eles são representados pelas abreviações:

- ECE: entrada da camada de entrada;
- SCE/ECI: Saída da camada de entrada/Entrada da camada de itens;
- SCI/ECS: Saída da camada de itens/Entrada da camada de saída;
- SCS/ECI: Saída da camada de saída/Entrada da camada de itens;
- SCS/ECE: Saída da camada de saída/Entrada da camada de entrada.

A camada de entrada é ativada por um dentre os seguintes valores:

- Zero: a execução da rede deve ser encerrada, o número máximo de épocas foi alcançado;
- Um: inicia a execução de mais uma iteração dentro da mesma época;
- Dois: uma nova época deve ser iniciada.

Uma vez que a camada de entrada não tenha determinado o encerramento da execução, o processamento continua na camada escondida que recebe, para cada nó, um sinal que indica se o item representado por tal nó é candidato a ser adicionado/retirado da mochila, ou seja, se o nó está ativo. Na primeira iteração de cada época, todos os nós estão ativos; a partir da segunda iteração, a camada recebe para cada nó um sinal proveniente da camada de saída que indica se o item correspondente foi escolhido na iteração anterior e, portanto, deve ser desativado na iteração corrente e nas subsequentes da época atual, ou se o item continua sendo candidato à adição/remoção.

A camada de saída, ou camada da mochila, utiliza a heurística para escolher dentre os itens ativos qual deve ser selecionado para adição ou remoção da mochila, levando em consideração as pseudoutilidades calculadas. Após a escolha do item, a camada de

saída determina se a busca pela solução da época atual se encerrou e, nesse caso, envia para a camada de entrada um sinal que indica que uma nova época deve ser iniciada ou, caso o número máximo de épocas tenha sido alcançado, que a execução deve ser encerrada. Caso a solução atual ainda não esteja completa, o sinal enviado para a camada de entrada é para que uma nova iteração seja realizada. Além disso, a camada de saída também envia n sinais, um para cada nó da camada escondida, especificando se os nós estão ativos ou inativos para a próxima iteração da época atual. Um nó é desativado quando o item por ele representado é adicionado/removido da mochila.

Assim como implementado por Deane e Agarwal [22, 23], a rede neural pode realizar retrocesso, caso haja uma sequência relativamente grande de épocas sem melhora na solução global obtida. O tamanho limite dessa sequência é especificado por um número inteiro fornecido como parâmetro de configuração da rede. Para possibilitar o retrocesso, o valor da melhor solução obtida até o momento, os itens nela contidos e os pesos utilizados na rede quando tal solução foi alcançada ficam armazenados em estruturas auxiliares; quando o número de épocas consecutivas cujas soluções são inferiores ou iguais a essa solução ótima alcança o limite especificado, o retrocesso é realizado: os valores armazenados são restaurados como solução corrente e uma nova época é iniciada.

Essa abordagem considera que, se após um determinado número de épocas não houve progresso, isso pode significar que naquela "vizinhança" há uma grande probabilidade de não ser mais possível melhorar a solução porque um ótimo local já foi alcançado naquela região da busca. A ideia é que a busca seja restaurada a seu melhor ponto e, a partir daí, siga em uma vizinhança diferente.

4.2 Implementação

As redes neurais aumentadas, baseadas na abordagem proposta por Deane e Agarwal [22, 23], foram implementadas em uma versão sequencial e duas paralelas, uma usando MPI e outra GPGPU. Nas implementações e resultados descritos nesta seção, utilizou-se a Equação (4.6) de atualização de pesos da abordagem híbrida proposta em [21] devido aos melhores resultados alcançados em comparação com a equação original descrita na Seção 4.1.

$$w_i^k = w_i^{k-1} + \alpha * gap * \beta_i, i = 1, \ldots, n \ . \tag{4.6}$$

Na equação alterada, removeu-se o fator que representa o valor do item, utilizado na versão original, dado que tal valor já é considerado no cálculo da pseudoutilidade. Com a utilização desse fator, o valor de um item era considerado duas vezes, o que levava itens com maior valor a serem favorecidos com relação aos de menor valor, reduzindo, como consequência, a variabilidade da rede.

Em todas as abordagens, fornece-se como entrada os parâmetros de configuração da rede: número de iterações, taxa de aprendizado e número de iterações para retrocesso. Os passos executados pela solução sequencial implementada utilizando a heurística ST e KMW são, respectivamente, ilustrados nos Algoritmos 4.1 e 4.2.

Algoritmo 4.1: RNA_ST(arq, *max*, α)

```
1  LêEntrada();
2  enquanto sinal de entrada for diferente de zero faça
3  |  se sinal for igual a um então
4  |  |  Passa para a próxima iteração da mesma época;
5  |  senão
6  |  |  Incrementa o contador de épocas;
7  |  |  Calcula o valor do gap da época anterior;
8  |  |  Coloca todos os itens na mochila;
9  |  |  Atualiza o vetor de pesos;
10 |  fim
11 |  Verifica nós ativos e calcula pseudoutilidades;
12 |  Retira item com a menor pseudoutilidade;
13 |  se a mochila for viável então
14 |  |  Encerra a época;
15 |  |  se número de épocas for igual a max então
16 |  |  |  Envia 0 para a camada de entrada;
17 |  |  senão
18 |  |  |  Envia 2 para a camada de entrada;
19 |  |  fim
20 |  senão
21 |  |  Envia 1 para a camada de entrada;
22 |  fim
23 fim
```

Algoritmo 4.2: RNA_KMW(arq, max, α)

```
1  LêEntrada();
2  enquanto sinal de entrada diferente de zero faça
3      se sinal for igual a um então
4          Passa para a próxima iteração da mesma época;
5      senão
6          Incrementa o contador de épocas;
7          Calcula o valor do gap da época anterior;
8          Retira todos os itens da mochila;
9          Atualiza o vetor de pesos;
10     fim
11     Verifica nós ativos e calcula pseudoutilidades;
12     Tenta adicionar algum item com a maior pseudoutilidade possível;
13     se nenhum item foi adicionado então
14         Encerra a época;
15         se número de épocas for igual a max então
16             Envia 0 para a camada de entrada;
17         senão
18             Envia 2 para a camada de entrada;
19         fim
20     senão
21         Envia 1 para a camada de entrada;
22     fim
23 fim
```

Para avaliar o comportamento das redes neurais aumentadas em um ambiente de memória compartilhada, projetaram-se implementações paralelas tendo em mente a arquitetura de GPUs. Para tal, foram considerados *grids* contendo apenas um bloco de *threads*, as dimensões do bloco variam de acordo com a etapa do algoritmo, assumindo uma entre duas possíveis configurações:

- Configuração 1: Um bloco unidimensional com n *threads*, onde n é o número de itens. Utilizada para executar em paralelo os cálculos que precisam ser realizados individualmente para cada item;
- Configuração 2: Um bloco unidimensional com m *threads*, onde m é o número de recursos. Utilizada para atividades que são executadas para cada recurso.

A CPU é responsável por ler o arquivo de entrada e por alocar memória para armazenar localmente as estruturas que representam a instância do problema (número de itens e recursos, valores dos itens, capacidades dos recursos e uso dos itens pelos recursos) e para as estruturas auxiliares, tais como o vetor de pesos. Todos os dados lidos são copiados para a memória global da GPU e, com base nessa configuração, as *threads* são coordenadas para trabalhar sobre uma parte desses dados.

Após a leitura dos dados, calculam-se os valores dos vetores cs ou rc, dependendo da heurística em questão. As etapas executadas em cada época são bastante similares à versão sequencial: a rede neural executa por um número predeterminado de épocas, cada qual composta por um número variável de iterações, o qual depende do progresso

da busca pela solução utilizando a heurística escolhida. A diferença fundamental com relação à implementação sequencial está no fato de que algumas atividades são executadas em paralelo pelas *threads* da GPU.

Na primeira iteração de cada época, o procedimento `ConfiguraÉpocaKernel()` realiza atividades para iniciar o processo de busca pela solução: adicionar ou retirar todos os itens da mochila e atualizar os pesos de acordo com o *gap* da solução da época anterior (ou atribuir a eles o valor 1, caso seja a primeira época). Esse é um procedimento executado na GPU e utiliza um bloco com a configuração 1. Outra etapa também executada na GPU é a inicialização dos vetores auxiliares utilizados no cálculo das pseudoutilidades dos itens, essa tarefa é realizada pelo procedimento `STIniciaKernel()` na implementação da heurística ST e pelo procedimento `KMWIniciaKernel()` na heurística KMW. Em ambos os procedimentos, a configuração 2 é utilizada na definição do bloco de *threads*.

A cada iteração, a função `CamadaEscondidaKernel()` (executada na GPU), que implementa a camada escondida e utiliza a configuração 1 de bloco, verifica quais itens ainda estão ativos e calcula suas pseudoutilidades. A camada de saída é executada na CPU e é responsável por decidir qual item deve ser adicionado/retirado da mochila ao final da iteração. Essa camada também é responsável por enviar para a camada de entrada o sinal que define se uma nova iteração deve ser realizada, se uma nova época deve ser iniciada ou, ainda, se o o número máximo de épocas foi alcançado e, portanto, o processamento deve ser encerrado. Os Algoritmos 4.3 e 4.4 ilustram os passos executados pelas implementações paralelas usando as heurísticas ST e KMW, respectivamente.

Algoritmo 4.3: RNA_ST_GPGPU(arq, α, *max*)

```
1  LêEntrada();
2  enquanto sinal de entrada for diferente de zero faça
3      se sinal for igual a um então
4          Passa para a próxima iteração da mesma época;
5      senão
6          ConfiguraÉpocaKernel();
7          STIniciaKernel();
8      fim
9      CamadaEscondidaKernel();
10     Retira item com a menor pseudoutilidade;
11     se a mochila for viável então
12         Encerra a época;
13         se número de épocas for igual a max então
14             Envia 0 para a camada de entrada;
15         senão
16             Envia 2 para a camada de entrada;
17         fim
18     senão
19         Envia 1 para a camada de entrada;
20     fim
21 fim
```

Algoritmo 4.4: RNA_KMW_GPGPU(arq, α, max)

```
1  LêEntrada();
2  enquanto sinal de entrada for diferente de zero faça
3      se sinal for igual a um então
4          Passa para a próxima iteração da mesma época;
5      senão
6          ConfiguraÉpocaKernel();
7          KMWIniciaKernel();
8      fim
9      CamadaEscondidaKernel();
10     Tenta adicionar algum item com a maior pseudoutilidade possível;
11     se nenhum item foi adicionado então
12         Encerra a época;
13         se número de épocas for igual a max então
14             Envia 0 para a camada de entrada;
15         senão
16             Envia 2 para a camada de entrada;
17         fim
18     senão
19         Envia 1 para a camada de entrada;
20     fim
21 fim
```

O comportamento da rede neural aumentada utilizando memória distribuída também foi testado, foram implementadas versões das duas heurísticas utilizando a biblioteca MPI (*Message Passing Interface*) [56]. Essa biblioteca possibilita que os programas desenvolvidos sejam executados em uma plataforma com memória distribuída e com comunicação por troca de mensagens através de uma rede de interconexão.

A estratégia adotada para paralelização usando MPI determina que a camada de entrada e a camada escondida sejam executadas em paralelo pelos p processadores e que a camada de saída seja executada pelo processador raiz com base nos dados resultantes do processamento paralelo. O processador raiz é responsável por iniciar o processamento realizando a leitura dos dados de entrada e os distribuindo para os demais. Cada processador recebe n/p elementos do vetor de valores dos itens, bem como parte da matriz de usos de recursos equivalentes a tais itens contendo $n/p * m$ elementos. O vetor de capacidades dos m recursos é replicado em todos os processadores.

Após a distribuição dos dados, a execução das épocas tem início e cada processador realiza as etapas da camada de entrada e da camada escondida considerando os itens armazenados localmente. Na primeira época, e sempre que o sinal de entrada for igual a 2, as configurações de início de época são executadas, ou seja, todos os itens são adicionados ou retirados da mochila e os pesos são atualizados de acordo com o *gap* da época anterior (exceto na primeira época). Além disso, dependendo da heurística em questão, um dos dois procedimentos seguintes é invocado:

- `STInicia()`: preenche um vetor com m posições que contém o quanto a capacidade de cada item é extrapolada considerando todos os itens presentes na mochila;

- `KMWInicia()`: inicia o vetor de capacidades residuais de recursos. Esse vetor com m posições armazena, a cada iteração, o quanto de cada recurso ainda está disponível, assim, nesse procedimento, ele recebe uma cópia do vetor de capacidades de recursos.

A cada iteração de uma época, os processadores são responsáveis por calcular as pseudoutilidades de todos os itens locais e, então, enviar para o processador raiz os valores calculados. A camada de saída é implementada no processador raiz que, após receber as pseudoutilidades locais, encontra o elemento adequado à adição/remoção baseado nos valores recebidos. O vetor solução é atualizado para refletir a adição/remoção do item escolhido e, por fim, a nova solução é distribuída para os demais processadores, o que possibilita que tal item seja desativado nas próximas iterações da época atual. Assim como nas outras implementações, verifica-se a viabilidade da solução e a contagem de épocas para enviar um sinal adequado (0, 1 ou 2) para os processadores continuarem o processamento da rede neural. Os procedimentos das Figuras 4.5 e 4.6 ilustram, respectivamente, as implementações paralelas com MPI das heurísticas ST e KMW.

Algoritmo 4.5: RNA_ST_MPI(arq, α, max, p)

```
1  Processador raiz lê arquivo de entrada;
2  Distribuição dos dados para os p processadores;
3  enquanto sinal de entrada for diferente de zero faça
4      se sinal for igual a um então
5          Passa para a próxima iteração da mesma época;
6      senão
7          Incrementa contador de épocas;
8          Cada processador calcula gap da época anterior;
9          Cada processador adiciona todos os itens locais à mochila;
10         Cada processador atualiza vetor de recursos disponíveis;
11         STInicia();
12     fim
13     Cada processador calcula pseudoutilidades dos itens locais;
14     Cada processador envia pseudoutilidades para raiz;
15     Processador raiz remove item com menor pseudoutilidade;
16     Processador raiz envia nova solução e sinais de ativação para os demais;
17     se mochila for viável então
18         Encerra a época;
19         se número de épocas for igual a max então
20             Envia 0 para a camada de entrada dos demais processadores;
21         senão
22             Envia 2 para a camada de entrada dos demais processadores;
23         fim
24     senão
25         Envia 1 para a camada de entrada dos demais processadores;
26     fim
27 fim
```

Algoritmo 4.6: RNA_KMW_MPI(arq, α, max, p)

```
1  Processador raiz lê arquivo de entrada;
2  Distribuição dos dados para os p processadores;
3  enquanto sinal de entrada for diferente de zero faça
4      se sinal for igual a um então
5          Passa para a próxima iteração da mesma época;
6      senão
7          Incrementa contador de épocas;
8          Cada processador calcula gap da época anterior;
9          Cada processador remove todos os itens locais da mochila;
10         Cada processador atualiza vetor de recursos disponíveis;
11         KMWInicia();
12     fim
13     Cada processador calcula pseudoutilidades dos itens locais;
14     Cada processador envia pseudoutilidades para raiz;
15     Processador raiz tenta adicionar item com maior pseudoutilidade possível;
16     Processador raiz envia nova solução e sinais de ativação para os demais;
17     se nenhum item foi adicionado então
18         Encerra a época;
19         se número de épocas for igual a max então
20             Envia 0 para a camada de entrada dos demais processadores;
21         senão
22             Envia 2 para a camada de entrada dos demais processadores;
23         fim
24     senão
25         Envia 1 para a camada de entrada dos demais processadores;
26     fim
27 fim
```

4.3 Resultados

Para avaliação das variantes implementadas, testes preliminares foram realizados e, pela análise dos resultados obtidos, percebeu-se que as implementações em MPI não levaram a resultados satisfatórios, pelas razões explicitadas no Capítulo 1. Da mesma maneira, a heurística KMW levou a soluções de maior qualidade do que a heurística ST tanto na versão sequencial quanto nas abordagens paralelas. Por esse motivo, os testes mais detalhados e validados abordados nesta seção foram focados apenas na versão sequencial e na paralela usando GPGPU com a heurística KMW.

A RNA recebe como parâmetros de entrada o número de épocas a serem realizadas, a taxa de aprendizado da rede e o valor limite para execução de retrocesso. As execuções dos testes foram, a princípio, realizadas considerando 1000 épocas tanto sequenciais quanto usando GPGPU. A qualidade dos resultados alcançados pela versão sequencial foram consideravelmente inferiores aos da versão paralela, a qual, entretanto, obteve *speedups* de no máximo 3,8%, sendo mais lenta com diversas instâncias, em especial, com as menores. Com base nesse comportamento, o programa sequencial foi executado novamente usando 10000 épocas, para avaliar a influência do aumento do número de

épocas na qualidade das soluções. Os resultados obtidos com a nova configuração foram melhores, porém demandaram um tempo de execução aproximadamente 10 vezes maior, o que já era esperado devido ao aumento da mesma escala no número de épocas, entretanto a qualidade das soluções não melhorou na mesma proporção. Observando essa situação, pode-se perceber que, a partir de um determinado número de épocas, não há uma melhora significativa na qualidade das soluções, não compensando o aumento no tempo de execução.

Foram testadas duas taxas de aprendizado para a rede neural, com os valores de 0,001 e 0,01, podendo-se constatar que a variação destas taxas também afetou a qualidade das soluções. Com a taxa de 0,001, os *gaps* das instâncias com mais de 500 itens foram menores, enquanto que para as demais instâncias, o uso da outra taxa foi mais vantajoso. Essa característica leva a crer que uma taxa de aprendizado reduzida não é suficiente para explorar adequadamente a vizinhança quando o número de épocas e/ou iterações da rede é reduzido, o que faz com que a busca fique "presa" em ótimos locais de baixa qualidade. Essa suposição é ratificada quando se faz a análise dos resultados obtidos pela implementação paralela com as duas diferentes taxas, nos quais a taxa de 0,01 levou a melhores resultados em todas as situações, dado que o número de épocas das execuções paralelas foi menor.

As Tabelas 4.1, 4.2 e 4.3 mostram os resultados de 10000 iterações da implementação sequencial de RNA.

Tabela 4.1: Resultados da implementação sequencial de RNA para as instâncias de SAC-94.

		RNA Sequencial				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	32	0,0000	0,2366	0,1816	0,7248
PB	6	150	0,0000	0,0887	0,0464	0,7441
PET	6	121	0,0000	0,0797	0,0257	0,7779
SENTO	2	60	0,0000	0,0000	0,0000	3,9226
WEING	2	210	0,0000	0,0007	0,0000	2,4175
WEISH	30	889	0,0000	0,0008	0,0015	2,0903

Tabela 4.2: Resultados da implementação sequencial de RNA para as instâncias de ORLIB agrupadas por configurações.

			RNA Sequencial				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	0	0,3158	0,8398	0,1640	1,8484
		0,50	0	0,0690	0,5609	0,1494	2,9863
		0,75	30	0,0000	0,3843	0,1349	4,0702
	250	0,25	0	0,1425	0,4619	0,1126	11,1218
		0,50	0	0,0590	0,2594	0,0538	18,4477
		0,75	0	0,0227	0,1961	0,0403	24,8154
	500	0,25	0	0,0364	0,2127	0,0510	45,5186
		0,50	0	0,0359	0,1289	0,0245	76,4738
		0,75	0	0,0372	0,0832	0,0201	101,4797
10	100	0,25	0	0,5395	1,6318	0,3551	2,4642
		0,50	0	0,4337	1,0364	0,1786	4,3163
		0,75	0	0,0395	0,5833	0,1100	6,0810
	250	0,25	0	0,2840	0,7682	0,1498	15,0291
		0,50	0	0,1671	0,5060	0,1014	26,8535
		0,75	0	0,0427	0,2631	0,0486	37,4315
	500	0,25	0	0,1135	0,3518	0,0701	61,1679
		0,50	0	0,0950	0,2187	0,0342	109,7134
		0,75	0	0,0613	0,1430	0,0138	152,0651
30	100	0,25	0	0,9067	2,2859	0,4067	4,8535
		0,50	0	0,6297	1,3984	0,2602	9,3616
		0,75	0	0,3578	0,9678	0,1456	13,9219
	250	0,25	0	0,9723	1,8067	0,2582	30,6991
		0,50	0	0,4692	0,9322	0,1109	59,3271
		0,75	0	0,2014	0,5464	0,0859	87,1541
	500	0,25	0	0,5642	1,1433	0,1213	125,8259
		0,50	0	0,3449	0,6124	0,0564	242,3274
		0,75	0	0,1826	0,3552	0,0259	352,5113

Tabela 4.3: Resultados da implementação sequencial de RNA para as instâncias de GK.

		RNA Sequencial				
m	n	OPT	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
15	100	0	0,5311	0,6444	0,0439	8,7956
25	100	0	0,8338	1,1403	0,1837	11,5195
25	150	0	0,7603	1,0431	0,1301	27,5434
50	150	0	0,6936	1,0196	0,1039	42,3335
25	200	0	0,5690	0,7666	0,1219	52,0517
50	200	0	0,7299	0,9771	0,1151	77,9059
25	500	0	0,1821	0,2335	0,0350	419,8423
50	500	0	0,4786	0,5229	0,0100	583,4163
25	1500	0	0,0430	0,0591	0,0110	6349,4797
50	1500	0	0,2025	0,2135	0,0031	7789,9975
100	2500	0	0,2783	0,3014	0,0008	36731,5515

Para alcançar bons resultados com a implementação usando GPGPU menos épocas foram necessárias. As Tabelas 4.4, 4.5 e 4.6 mostram os resultados e *speedups* obtidos executando 1000 épocas da RNA paralela.

Tabela 4.4: Resultados da implementação GPGPU de RNA para as instâncias de SAC-94.

		RNA GPGPU					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	1	0,0000	1,2347	0,5036	1,0669	0,6761
PB	6	36	0,0000	1,0450	0,5604	0,8369	0,8783
PET	6	98	0,0000	0,3027	0,1943	0,9129	0,6695
SENTO	2	18	0,0000	0,0764	0,0636	1,5812	2,4625
WEING	2	116	0,0000	0,0875	0,0842	1,0795	1,1411
WEISH	30	627	0,0000	0,0571	0,0533	1,2764	1,4297

Tabela 4.5: Resultados da implementação GPGPU de RNA para as instâncias de ORLIB agrupadas por configurações.

			RNA GPGPU					
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	5	0,0000	0,5530	0,2108	1,3206	1,4000
		0,50	1	0,0000	0,3330	0,1094	2,3304	1,2815
		0,75	32	0,0000	0,1966	0,0666	3,3096	1,2298
	250	0,25	0	0,0521	0,3054	0,0879	3,3435	3,3269
		0,50	0	0,0234	0,1658	0,0455	5,8041	3,1786
		0,75	0	0,0227	0,1278	0,0347	8,2822	2,9964
	500	0,25	0	0,0157	0,1480	0,0376	7,2106	6,3131
		0,50	0	0,0063	0,1529	0,2306	12,3575	6,1886
		0,75	0	0,0237	0,0580	0,0134	17,4114	5,2438
10	100	0,25	4	0,0000	1,1742	0,3132	1,2769	1,9299
		0,50	1	0,0000	0,6720	0,1807	2,3846	1,8103
		0,75	1	0,0000	0,3678	0,1046	3,4377	1,7690
	250	0,25	0	0,2433	0,6237	0,1160	3,2884	4,5705
		0,50	0	0,0793	0,3683	0,0736	6,0273	4,4555
		0,75	0	0,0427	0,1937	0,0492	8,6926	4,3062
	500	0,25	0	0,0842	0,2772	0,0522	7,0885	8,6296
		0,50	0	0,0495	0,1679	0,0299	12,7260	8,6215
		0,75	0	0,0390	0,1133	0,0206	18,2509	8,3320
30	100	0,25	0	0,3098	1,8847	0,4023	1,4108	3,4404
		0,50	0	0,2630	0,9706	0,2298	2,7785	3,3694
		0,75	6	0,0000	0,5419	0,1713	4,1613	3,3457
	250	0,25	0	0,7963	1,5727	0,2335	3,7306	8,2295
		0,50	0	0,3259	0,7753	0,1050	7,1794	8,2637
		0,75	0	0,2014	0,4159	0,0608	10,6372	8,1934
	500	0,25	0	0,4510	1,0400	0,1214	8,1924	15,3591
		0,50	0	0,2639	0,5124	0,0512	15,3773	15,7589
		0,75	0	0,1695	0,2984	0,0367	22,4255	15,7193

Tabela 4.6: Resultados da implementação GPGPU de RNA para as instâncias de GK.

		RNA GPGPU					
m	n	OPT	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
15	100	0	0,3717	0,6205	0,0753	2,4894	3,5332
25	100	0	0,5811	1,0241	0,1907	2,7272	4,2239
25	150	0	0,7072	0,9889	0,1711	4,1436	6,6472
50	150	0	0,7283	1,0057	0,1096	5,1283	8,2549
25	200	0	0,4234	0,7031	0,1578	5,5348	9,4044
50	200	0	0,5344	0,9428	0,1401	6,8042	11,4497
25	500	0	0,1561	0,1990	0,0249	14,8004	28,3670
50	500	0	0,4254	0,5117	0,0272	18,2666	31,9390
25	1500	0	0,0430	0,0541	0,0076	55,0975	115,2408
50	1500	0	0,1920	0,2125	0,0054	71,6803	108,6770
100	2500	0	0,2447	0,2884	0,0171	189,1680	194,1742

Os valores apresentados nas tabelas nos mostram que os *gaps* alcançados pela versão GPGPU foram menores que os da versão sequencial em quase todas as instâncias, com exceção das instâncias de SAC-94, as menores testadas. Devido ao tamanho reduzido, presume-se que essas instâncias não conseguem escapar dos máximos locais de baixa qualidade com as poucas épocas executadas pela execução paralela, o que ocorre com maior facilidade com a versão sequencial que executa um número de épocas 10 vezes maior.

Como forma de ressaltar a qualidade das implementações de RNA propostas, os *gaps* obtidos para as instâncias de ORLIB foram comparados aos da rede neural aumentada descrita em [22] e com a implementação da heurística KMW pura; não foi possível comparar os tempos de execução, pois tais informações não estão disponíveis nas referências consultadas. A Tabela 4.7 apresenta os *gaps* agrupados por número de itens obtidos pelas quatro abordagens, destacando em negrito o melhor *gap* de cada configuração.

Tabela 4.7: *Gaps* comparados aos da RNA de Deane e Agarwal e da heurística KMW pura.

m	n	RNA Sequencial	RNA GPGPU	RNA-KMW	KMW-Pura
5	100	0,59	**0,36**	0,65	1,43
5	250	0,31	**0,20**	0,21	0,55
5	500	0,14	0,12	**0,09**	0,24
10	100	1,08	**0,74**	1,22	2,79
10	250	0,51	**0,40**	0,50	1,13
10	500	0,24	**0,19**	0,22	0,44
30	100	1,55	**1,13**	2,28	4,37
30	250	1,10	**0,92**	1,23	2,08
30	500	0,70	**0,62**	0,74	1,13

Pode-se verificar que em todas as configurações, exceto na configuração 5×500, o programa usando GPGPU obteve resultados melhores do que os das demais abordagens, enquanto a versão sequencial apresentou algus resultados melhores e outros de qualidade inferior. Essa situação comprova que a implementação GPGPU da RNA proposta é viável para a solução do problema da mochila multidimensional.

A validação com a aplicação do teste de Wilcoxon de soma de *ranks* mostrou que os resultados obtidos para todos os conjuntos de testes não são significativamente diferentes dos valores de referência. Os p-valores computados para instâncias de SAC-94 foram 0,4865 e 0,4450, respectivamente para as versões sequencial e paralela; o valor de W, para as instâncias de GK, foi igual a 121, tanto para a versão sequencial quanto para a paralela. A Tabela 4.8 mostra os p-valores calculados para as configurações das instâncias de ORLIB.

Tabela 4.8: p-valores calculados para os resultados da RNA para cada configuração de testes de ORLIB.

		p-valor	
m	n	Sequencial	GPGPU
5	100	0,3367	0,3753
	250	0,3395	0,3558
	500	0,3669	0,3725
10	100	0,2390	0,2821
	250	0,2673	0,3024
	500	0,3181	0,3503
30	100	0,1610	0,2210
	250	0,1044	0,1185
	500	0,1469	0,1646

4.4 Contribuições

As implementações de redes neurais aumentadas apresentadas neste capítulo, levaram a contribuições importantes, tais como:

- Implementação de um algoritmo paralelo para RNA;
- Publicação de artigo em *XL Latin American Computing Conference* (CLEI 2014) [18];
- Proposta de uma nova equação de atualização de pesos, que levou a resultados de melhor qualidade;
- Percepção de que características gulosas e de aprendizado tornam o algoritmo bastante eficiente para instâncias menores.

CAPÍTULO 5

GRASP

5.1 Fundamentação

Um grande número de metaheurísticas se baseiam na ideia de encontrar uma solução inicial e de explorar sua vizinhança em busca de soluções de melhor qualidade; esses são os chamados *métodos de busca*. Tais métodos, em geral, precisam fazer uso de alguma técnica para evitar que a busca seja limitada por ótimos locais de baixa qualidade encontrados ao longo do processo, os quais podem impedir que ótimos globais, o objetivo da otimização, sejam alcançados. Os *métodos de múltiplos inícios (multi-start methods)* tentam suplantar essa limitação com o reinício do processo de busca em outra região, assim que a atual tiver sido suficientemente explorada [50].

GRASP (*Greedy Randomized Adaptive Search Procedure*) é uma metaheurística de múltiplos inícios que realiza uma série de iterações compostas por duas fases: construção e busca local. A fase de construção é responsável por obter uma solução inicial válida para o problema; a busca local explora a vizinhança de tal solução até encontrar um ótimo local. O resultado do processo é a melhor solução encontrada ao longo das iterações. O Algoritmo 5.1 ilustra os passos executados pelo GRASP [60].

Algoritmo 5.1: GRASP(*maxIterações*, *semente*)

```
1 LêEntrada();
2 para k ← 1, maxIterações faça
3 |   Solução ← ConstruirSoluçãoAleatória(semente);
4 |   Solução ← BuscaLocal(Solução);
5 |   AtualizarSolução(Solução, MelhorSolução);
6 fim
7 retorna MelhorSolução;
```

O procedimento `ConstruirSoluçãoAleatória()` é responsável pela construção da solução inicial utilizando uma estratégia gulosa. Um conceito chave associado a essa etapa é a *lista de candidatos restrita* (*restricted candidate list* - RCL) que é composta pelos elementos que mais contribuem para melhorar a solução sendo construída, sem torná-la inviável. A escolha de tais elementos é realizada com o auxílio de uma

função gulosa de avaliação que garante que a RCL contenha os elementos que mais contribuem para maximizar ou minimizar os custos incrementais da solução, dependendo se o problema em questão é de maximização ou minimização, respectivamente.

O número de elementos contidos na RCL pode ser limitado de duas possíveis maneiras [60], baseado em cardinalidade ou em qualidade. Na primeira opção, a lista é construída com um número predeterminado p dos melhores elementos disponíveis. A segunda opção considera um parâmetro limiar $\alpha \in [0, 1]$ que determina quais elementos devem ser adicionados à lista; no caso de um problema de maximização, todo elemento disponível e cuja qualidade $c(e)$ pertença ao intervalo $[c^{max} - \alpha(c^{max} - c^{min}), c^{max}]$, que não torne a solução inviável, é adicionado à RCL. Os elementos são escolhidos aleatoriamente da RCL para adição à solução, assim, o parâmetro α controla o quão gulosa ou aleatória é a execução do algoritmo; quando $\alpha = 0$, o algoritmo é puramente guloso, quando $\alpha = 1$, ele funciona de maneira totalmente aleatória. A qualidade da solução inicial fornecida possui grande influência sobre a eficácia do procedimento de busca local, bem como diversos outros fatores, tais como a estrutura da vizinhança, a técnica de exploração utilizada e o quão rapidamente pode ser realizada a avaliação da função de custo [60].

Uma grande variedade de problemas têm sido resolvidos com a utilização de GRASP, tais como problemas em grafos, de roteamento, de alocação, entre outros. Li, Pardalos e Resende [45] apresentaram uma implementação de GRASP para o problema da alocação quadrática (*quadratic assignment problem* – QAP) utilizando como conjunto de testes as instâncias da biblioteca QAPLIB [12]. Resende e Ribeiro [63] descreveram uma aplicação de GRASP para o problema da planarização de grafos e mostraram que a abordagem proposta alcançou resultados melhores ou tão bons quanto aos das heurísticas previamente utilizadas. Martins, Ribeiro e Souza [49] propuseram uma implementação paralela de GRASP para o problema de árvores de Steiner em grafos que considera duas opções para a busca local, uma com vizinhança baseada em nós e outra baseada em caminhos; após a análise de ambas, os autores propuseram uma estratégia híbrida utilizando características das duas abordagens, a qual alcançou resultados no máximo quatro porcento inferiores às soluções conhecidas.

Em sua formulação básica, GRASP é uma metaheurística que não possui nenhum mecanismo de memória, na qual cada iteração é responsável por construir uma nova solução desde o começo, sem tirar proveito de informações provenientes de iterações anteriores, a incorporação da técnica *path-relinking* ao GRASP visa superar essa limitação. *Path-relinking* (PR) é uma estratégia de intensificação, originalmente proposta por Glover [33] que explora trajetórias que conectam soluções de elite obtidas por busca tabu ou busca dispersa [61]. A ideia é tentar incorporar atributos de soluções de alta qualidade obtidas anteriormente à solução em construção através da exploração de caminhos que conectam tais soluções de elite. O Algoritmo 5.2 ([61]) ilustra os passos do procedimento de *path-relinking* que explora um caminho entre uma solução inicial `ISolução` e uma solução alvo `ASolução`.

Algoritmo 5.2: PathRelinking(*ISolução*, *ASolução*)

```
1  movConjto ← conjunto de movimentos que transformam ISolução em ASolução;
2  MelhorSoluçãoValor ← max{f(ISolução), f(ASolução)};
3  MelhorSolução ← argmax{f(ISolução), f(ASolução)};
4  Solução ← ISolução;
5  enquanto movConjto ≠ ∅ faça
6  |  m* ← argmax{f(Solução ∪ {m}) : m ∈ movConjto};
7  |  movConjto ← movConjto ∖{m*};
8  |  Solução ← Solução ∪{m*};
9  |  se f(Solução) > MelhorSoluçãoValor então
10 |  |  MelhorSoluçãoValor ← f(Solução);
11 |  |  MelhorSolução ← Solução;
12 |  fim
13 fim
14 return MelhorSolução;
```

O algoritmo de *path-relinking* inicialmente cria um conjunto contendo os movimentos necessários para transformar a solução inicial na solução alvo e, a partir desse ponto, a cada iteração, seleciona o movimento m^* pertencente ao conjunto cuja adição maximiza o custo da nova solução. Após a seleção de m^*, a solução atual e, possivelmente a melhor solução, é atualizada para conter o novo item. O ciclo de iterações se encerra quando não houver mais elemento restante no conjunto, ou seja, quando a solução alvo tiver sido atingida [61].

De acordo com Resende e Ribeiro [61], a associação de *path-relinking* a GRASP melhora a qualidade das soluções obtidas e o tempo de execução em comparação com o procedimento básico de GRASP. A ideia foi inicialmente proposta por Laguna e Martí[43] e, desde então, tem sido aplicada a diversos problemas como, por exemplo, o problema de atribuição de três índices (*three-index assignment problem*) [5], o problema de Steiner em grafos [64], o problema das p-medianas [62], o problema de projeto de redes de dois caminhos (2-*path network design problem*) [65] e o problema do dimensionamento de lotes capacitado por múltiplas plantas [53]. Uma bibliografia mais ampla sobre GRASP e suas aplicações pode ser encontrada nos trabalhos de Festa e Resende [26, 27].

5.2 Implementação

Nesta seção são apresentadas as implementações sequenciais e paralelas de GRASP sem e com a associação de intensificação com *path-relinking*. As implementações sequenciais foram projetadas seguindo os passos básicos propostos por Resende e Ribeiro [60]; a RCL é construída usando a política baseada em qualidade. Para avaliar a qualidade dos itens, calcula-se a pseudoutilidade de cada item com base em seu valor e em sua demanda por cada recurso, como feito na implementação de redes neurais aumentadas com a heurística KMW; A RCL deve conter o itens com os maiores valores de pseudoutilidade. Os Algoritmos 5.3 e 5.4 ilustram a função principal e a função de construção da solução inicial da abordagem implementada, nos quais o parâmetro α é usado para guiar a construção da RCL.

Algoritmo 5.3: GRASP_MKP(*maxIterações*, α)

```
1  LêEntrada();
2  MelhorSolução ← ∅;
3  para k ← 1, maxIterações faça
4  |  Solução ← ∅;
5  |  Solução ← ConstróiSolução(Solução, α);
6  |  Solução ← BuscaLocal(Solução, α);
7  |  ValorSolução ← valor(Solução);
8  |  se Solução é melhor que MelhorSolução então
9  |  |  MelhorSolução ← Solução;
10 |  fim
11 fim
12 retorna MelhorSolução;
```

Algoritmo 5.4: ConstróiSolução(*Solução*, α)

```
1  repita
2  |  Calcula as pseudoutilidades dos itens;
3  |  Constrói a RCL;
4  |  Seleciona aleatoriamente um item e da RCL;
5  |  se e pode ser adicionado à Solução então
6  |  |  Solução ← Solução ∪{e};
7  |  fim
8  até que e não possa ser adicionado à Solução;
9  retorna Solução;
```

A busca local, implementada usando as ideias apresentadas em [73], trabalha de maneira iterativa para encontrar uma solução de melhor qualidade. O procedimento repete o processo de remover alguns itens da solução corrente até que qualquer item disponível possa ser incorporado a ela e, a partir daí, uma nova solução é construída. Os itens são removidos em ordem não-decrescente de pseudoutilidade. Um vetor lógico de n posições (denominado *marcado*) é usado para guiar o processo, "marcando" qual foi o primeiro item removido a cada iteração. Inicialmente, cada posição $i, 0 \leq i < n$ recebe os valores `falso` ou `verdadeiro`, respectivamente, caso o item esteja ou não presente na solução. As tentativas de obter uma solução melhor são repetidas até todos os itens estarem marcados. Os passos da busca local, baseados em [73], são listados no Algoritmo 5.5.

Algoritmo 5.5: BuscaLocal(*Solução*, α)

```
1  Inicializa o vetor marcado;
2  Copia Solução para SolCorrente;
3  enquanto houver algum item não marcado faça
4  |  Calcula as pseudoutilidades dos itens;
5  |  repita
6  |  |  Encontra o item e ∈ SolCorrente com a menor pseudoutilidade e o
   |  |  remove de SolCorrente;
7  |  até todos os itens disponíveis poderem ser adicionados a SolCorrente;
8  |  ConstróiSolução(SolCorrente, α);
9  |  se SolCorrente é melhor que Solução então
10 |  |  Solução ← SolCorrente;
11 |  |  Atualiza o vetor marcado;
12 |  senão
13 |  |  SolCorrente ← Solução;
14 |  |  Primeiro item removido é marcado;
15 |  fim
16 fim
17 retorna Solução;
```

A implementação paralela usando GPGPU visa expandir o espaço de busca na vizinhança com o uso de diferentes *threads* para, de maneira iterativa, construir diferentes soluções iniciais em paralelo e, então, executar a busca local em suas soluções locais. São utilizadas t *threads* e cada uma executa $1/t$ iterações do total especificado como entrada do programa. Esta abordagem pode ser vista como múltiplas execuções paralelas do programa sequencial; a CPU é utilizada para gerenciar as iterações e para encontrar a melhor solução obtida ao final de todo o processo. As funções responsáveis pela construção da solução inicial e pela busca local em cada *thread* (`KernelConstróiSolução()` e `KernelBuscaLocal()`, respectivamente) realizam as mesmas operações dos procedimentos sequenciais equivalentes. O Algoritmo 5.6 ilustra os passos da função principal de GRASP usando GPGPU.

Algoritmo 5.6: GRASP_MKP_GPGPU(*maxIterações*, α, t)

```
1  LêEntrada();
2  ValorMelhorSolução ← 0;
3  MelhorSolução ← ∅;
4  para k ← 1, maxIterações /t faça
5  |  KernelConstróiSolução(SoluçãoThread, α);
6  |  KernelBuscaLocal(SoluçãoThread, α);
7  |  CPU atribui a melhor solução encontrada pelas threads a Solução;
8  |  ValorSolução ← valor(Solução);
9  |  se ValorSolução > ValorMelhorSolução então
10 |  |  MelhorSolução ← Solução;
11 |  |  ValorMelhorSolução ← ValorSolução;
12 |  fim
13 fim
14 retorna (MelhorSolução, ValorMelhorSolução);
```

Com base nos resultados promissores obtidos com o uso de *path-relinking* associado ao funcionamento básico de GRASP, constatado nos trabalhos citados na Seção 5.1, e com o intuito de avaliar se tais melhorias também ocorrem quando o foco é o problema da mochila multidimensional, foram implementadas versões usando *path-relinking* dos dois algoritmos descritos até o momento. O Algoritmo 5.7 mostra os passos da implementação sequencial de GRASP com *path-relinking*.

Algoritmo 5.7: PR_GRASP_MKP(*maxIterações*, α)

```
1  LêEntrada();
2  MelhorSolução ← ∅;
3  para k ← 1, maxIterações faça
4  |  Solução ← ∅;
5  |  Solução ← ConstróiSolução(Solução, α);
6  |  Solução ← BuscaLocal(Solução, α);
7  |  se k > 1 então
8  |  |  Solução ← PathRelinking(Solução, MelhorSolução);
9  |  fim
10 |  se Solução é melhor que MelhorSolução então
11 |  |  MelhorSolução ← Solução;
12 |  fim
13 fim
14 retorna MelhorSolução;
```

Os passos do algoritmo são idênticos aos da versão básica de GRASP, à exceção das linhas entre 7 e 9, responsáveis pela chamada ao procedimento que executa o *path-relinking*. A fase de intensificação *path-relinking* implementada seguiu a mesma estratégia descrita por Arin e Rabaldi [7] na qual, a cada iteração, constrói-se um caminho que conecta a solução corrente com a melhor solução encontrada até o momento no processo. Diferentemente da opção descrita pelos autores, entretanto, a melhor entre as duas é utilizada como solução inicial e a outra é utilizada como solução alvo. Inicialmente, o algoritmo identifica os itens que pertencem exclusivamente a uma das soluções e, sistematicamente, troca os valores da solução inicial um a um, visando transformá-la na solução alvo. A cada passo dessa transformação, os vizinhos da solução corrente são as soluções potenciais que diferem em apenas uma posição – dentre as posições previamente identificadas –, então o vizinho que representa a solução com maior valor associado é escolhido para ser a nova solução no caminho. Ao término do processo, a solução corrente da iteração GRASP é atualizada para a melhor solução encontrada ao longo do caminho percorrido. O Algoritmo 5.8 ilustra os passos do *path-relinking*.

A versão usando GPGPU, assim como a sequencial, apenas teve o trecho de código contendo a chamada ao procedimento de *path-relinking* (`KernelPathRelinking()`) adicionado. Este novo procedimento também é executado em paralelo sobre a melhor solução encontrada globalmente e as respectivas soluções correntes locais e é implementado como a versão sequencial. O Algoritmo 5.9 mostra os passos executados pela implementação paralela de GRASP com *path-relinking*.

Algoritmo 5.8: PathRelinking(*Solução*, *MelhorSolução*)

```
1  se MelhorSolução é melhor que Solução então
2  |  prSol ← MelhorSolução;
3  |  valorPrSol ← valor(MelhorSolução);
4  senão
5  |  prSol ← Solução;
6  |  valorPrSol ← valor(Solução);
7  fim
8  Constrói vetor índices com os itens que pertencem a uma única solução;
9  numItems ← tamanho(índices);
10 valorMelhorPr ← 0;
11 melhorPrSol ← ∅;
12 para k ← 1, numItems faça
13 |  prSol ← melhor vizinho que difere em apenas um índice i ∈ índices de prSol;
14 |  valorPrSol ← valor(prSol);
15 |  se valorPrSol > valorMelhorPr então
16 |  |  melhorPrSol ← prSol;
17 |  |  valorMelhorPr ← valorPrSol;
18 |  fim
19 fim
20 se valorMelhorPr > valor(Solução) então
21 |  Solução ← melhorPrSol;
22 fim
23 retorna Solução;
```

Algoritmo 5.9: PR_GRASP_MKP_GPGPU(*maxIterações*, α, t)

```
1  LêEntrada();
2  ValorMelhorSolução ← 0;
3  MelhorSolução ← ∅;
4  para k ← 1, maxIterações /t faça
5  |  KernelConstróiSolução(SoluçãoThread, α);
6  |  KernelBuscaLocal(SoluçãoThread, α);
7  |  se k > 1 então
8  |  |  KernelPathRelinking(SoluçãoThread, MelhorSolução);
9  |  fim
10 |  CPU atribui a melhor solução encontrada pelas threads a Solução;
11 |  ValorSolução ← valor(Solução);
12 |  se ValorSolução > ValorMelhorSolução então
13 |  |  MelhorSolução ← Solução;
14 |  |  ValorMelhorSolução ← ValorSolução;
15 |  fim
16 fim
17 retorna (MelhorSolução, ValorMelhorSolução);
```

5.3 Resultados

Nesta seção são apresentados os resultados obtidos pelas implementações dos algoritmos descritos na Seção 5.2. A ideia de associar o *path-relinking* ao funcionamento básico de GRASP se mostrou um alternativa eficaz, levando a obtenção de soluções de melhor qualidade. O mecanismo de "memória" da melhor solução proporcionada pelo *path-relinking* possibilitou que características das melhores soluções encontradas ao longo do processo fossem mantidas pelas soluções subsequentes. No caso da implementação paralela, o uso da técnica também possibilitou que as melhores soluções transmitissem suas características para as construídas paralelamente pelas diferentes *threads*, fazendo com que elas explorassem diferentes vizinhanças das soluções de elite.

Após a análise de diferentes opções para o valor limiar para a construção da RCL, optou-se por usar o valor 0,1, o que corresponde a fazer com que a RCL contenha itens cuja qualidade é, no máximo, 10% inferior a do(s) item(ns) de melhor qualidade. Cada execução dos programas consistiu em 1000 iterações. As Tabelas 5.1, 5.2 e 5.3 apresentam os resultados médios obtidos pela implementação sequencial de GRASP sem o uso da intensificação com *path-relinking* para os três conjuntos de testes.

Tabela 5.1: Resultados da implementação sequencial de GRASP sem uso de *path-relinking* para as instâncias de SAC-94.

		GRASP sequencial				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	0	2,9257	3,2833	0,0000	0,2012
PB	6	0	1,7298	5,0695	0,0214	0,1666
PET	6	0	0,2491	3,3763	0,0164	0,2209
SENTO	2	0	0,2981	0,9082	0,0000	1,7340
WEING	8	0	0,0085	2,3660	0,0000	0,6613
WEISH	30	150	0,0000	0,6904	0,0033	0,7051

Tabela 5.2: Resultados da implementação sequencial de GRASP sem uso de *path-relinking* para as instâncias de ORLIB agrupadas por configurações.

			GRASP sequencial				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	59	0,0000	0,1589	0,0458	0,9664
		0,50	141	0,0000	0,0461	0,0186	1,5377
		0,75	117	0,0000	0,0360	0,0091	1,4806
	250	0,25	11	0,0000	0,1417	0,0452	6,2251
		0,50	2	0,0000	0,0699	0,0192	6,0982
		0,75	6	0,0000	0,0397	0,0168	4,7417
	500	0,25	1	0,0000	0,1579	0,0294	16,8531
		0,50	0	0,0009	0,0648	0,0131	16,1215
		0,75	0	0,0133	0,0445	0,0086	12,9989
10	100	0,25	41	0,0000	0,4955	0,1125	1,3664
		0,50	60	0,0000	0,1995	0,0502	2,1983
		0,75	167	0,0000	0,0558	0,0285	2,0708
	250	0,25	0	0,0084	0,3176	0,0618	9,2437
		0,50	0	0,0496	0,1966	0,0329	8,7584
		0,75	12	0,0000	0,0860	0,0229	6,7551
	500	0,25	0	0,0732	0,2475	0,0405	23,4826
		0,50	0	0,0529	0,1602	0,0207	22,0385
		0,75	0	0,0112	0,0910	0,0135	18,0649
30	100	0,25	3	0,0000	0,9842	0,2471	2,9125
		0,50	0	0,0872	0,4853	0,0899	5,0092
		0,75	0	0,0500	0,2721	0,0280	4,1209
	250	0,25	0	0,3462	0,8665	0,1340	20,4392
		0,50	0	0,2181	0,4917	0,0627	20,3288
		0,75	0	0,1031	0,2495	0,0359	13,9930
	500	0,25	0	0,3101	0,8025	0,0698	49,9635
		0,50	0	0,1496	0,3819	0,0315	47,8356
		0,75	0	0,0705	0,2346	0,0219	38,4013

Tabela 5.3: Resultados da implementação sequencial de GRASP sem uso de *path-relinking* para as instâncias de GK.

		GRASP sequencial				
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
115	100	0	0,1859	0,2452	0,0272	2,1534
25	100	0	0,4800	0,8607	0,1205	3,6371
25	150	0	0,9901	1,0826	0,0847	8,0688
50	150	0	0,5202	0,5893	0,0637	14,2126
25	200	0	0,0397	0,2064	0,0478	15,0124
50	200	0	1,1731	1,4038	0,0833	23,3570
25	500	0	0,2810	0,3608	0,0307	89,0985
50	500	0	0,2712	0,2952	0,0201	160,2657
25	1500	0	1,1087	1,1746	0,0230	387,5510
50	1500	0	1,3841	1,4397	0,0265	674,3600
100	2500	0	0,2132	0,2474	0,0152	2646,9373

Dado que o número de iterações é subdividido entre as *threads*, o número de *threads* nas execuções do programa GPGPU foi configurado como 100; se um número maior de *threads* fosse utilizado, muito poucas iterações seriam executadas e, consequentemente, a qualidade das soluções seria prejudicada. As *threads* foram organizadas em 25 blocos; tal configuração levou aos melhores resultados comparada com as demais testadas para subdividir as 100 *threads* (1, 2, 4, 5, 10, 20, 50 e 100 blocos). Estabeleceu-se também um limite de 20 iterações da fase de busca local, visto que o aumento da qualidade da solução com número maior de iterações não é tão significativo quanto o aumento no tempo de execução acarretado pelas iterações adicionais. As Tabelas 5.4, 5.5 e 5.6 apresentam os resultados médios e os *speedups* obtidos pela implementação paralela de GRASP sem o uso da intensificação com *path-relinking* para os três conjuntos de testes.

Tabela 5.4: Resultados da implementação usando GPGPU de GRASP sem uso de *path-relinking* para as instâncias de SAC-94.

		GRASP GPGPU					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	0	1,9310	3,0539	0,3423	0,8330	0,2415
PB	6	0	1,7298	4,9613	0,1679	0,9252	0,1800
PET	6	0	0,2491	3,3413	0,0085	0,5339	0,4137
SENTO	2	0	0,2981	0,9082	0,0000	3,3269	0,5212
WEING	8	0	0,0085	2,3660	0,0000	0,7082	0,9338
WEISH	30	150	0,0000	0,6254	0,0021	1,3656	0,5163

Tabela 5.5: Resultados da implementação usando GPGPU de GRASP sem uso de *path-relinking* para as instâncias de ORLIB agrupadas por configurações.

			GRASP GPGPU					
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	65	0,0000	0,1769	0,0549	1,8334	0,5270
		0,50	98	0,0000	0,0694	0,0248	1,5998	0,9614
		0,75	106	0,0000	0,0791	0,0110	1,3722	1,0797
	250	0,25	5	0,0000	0,1521	0,0461	5,4413	1,1419
		0,50	0	0,0027	0,0791	0,0228	4,9409	1,2332
		0,75	5	0,0000	0,0456	0,0164	4,6667	1,0147
	500	0,25	1	0,0000	0,1695	0,0317	12,8413	1,3134
		0,50	0	0,0076	0,0692	0,0147	13,3407	1,2091
		0,75	0	0,0063	0,0476	0,0099	13,6826	0,9508
10	100	0,25	34	0,0000	0,5455	0,1141	2,6698	0,5114
		0,50	46	0,0000	0,2756	0,0529	2,2418	0,9822
		0,75	87	0,0000	0,1341	0,0363	1,8078	1,1517
	250	0,25	0	0,0706	0,3874	0,0793	7,7056	1,1995
		0,50	1	0,0000	0,2646	0,0426	6,9348	1,2638
		0,75	0	0,0300	0,1146	0,0183	6,1278	1,1023
	500	0,25	0	0,0895	0,2718	0,0434	17,8988	1,3126
		0,50	0	0,0939	0,1792	0,0238	18,3763	1,1987
		0,75	0	0,0112	0,1041	0,0155	18,0979	0,9984
30	100	0,25	2	0,0000	1,0464	0,2583	5,6221	0,5195
		0,50	0	0,1291	0,6381	0,1170	4,7454	1,0557
		0,75	0	0,1036	0,4016	0,0676	3,5138	1,1731
	250	0,25	0	0,5332	1,1520	0,1338	16,5650	1,2341
		0,50	0	0,2580	0,6211	0,0649	15,1510	1,3413
		0,75	0	0,1770	0,3122	0,0398	12,6207	1,1091
	500	0,25	0	0,4458	0,9632	0,0679	39,2471	1,2726
		0,50	0	0,1810	0,4432	0,0346	40,2977	1,1874
		0,75	0	0,1591	0,2725	0,0226	37,9511	1,0117

Tabela 5.6: Resultados da implementação usando GPGPU de GRASP sem uso de *path-relinking* para as instâncias de GK.

		GRASP GPGPU					
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
115	100	0	0,2124	0,2567	0,0198	2,2903	0,9402
25	100	0	0,8085	1,1445	0,1579	3,2597	1,1158
25	150	0	1,3437	1,4360	0,1185	5,6600	1,4256
50	150	0	0,6242	0,8738	0,0994	9,3919	1,5133
25	200	0	0,4102	0,4918	0,0378	8,5644	1,7529
50	200	0	1,5250	1,7827	0,0988	14,5508	1,6052
25	500	0	0,3591	0,4699	0,0485	33,0904	2,6926
50	500	0	0,4254	0,4608	0,0253	54,2252	2,9556
25	1500	0	1,3566	1,4297	0,0326	219,5180	1,7656
50	1500	0	1,6686	1,7711	0,0329	410,3496	1,6434
100	2500	0	0,3161	0,3610	0,0289	1824,2332	1,4510

Os resultados obtidos pelas implementação sequencial de GRASP usando *path-relinking* para os conjuntos de teste são apresentados nas Tabelas 5.7, 5.8 e 5.9, enquanto as Tabelas 5.10, 5.11 e 5.12 mostram os resultados da versão usando GPGPU e os *speedups* alcançados.

Tabela 5.7: Resultados da implementação sequencial de GRASP com uso de *path-relinking* para as instâncias de SAC-94.

		GRASP sequencial com PR				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	0	2,9257	3,2833	0,0000	0,1993
PB	6	0	1,4438	5,0495	0,0816	0,1670
PET	6	0	0,2491	3,3560	0,0187	0,2206
SENTO	2	0	0,2981	0,9082	0,0000	1,7339
WEING	8	0	0,0085	2,3660	0,0000	0,6612
WEISH	30	150	0,0000	0,6885	0,0125	0,7092

Tabela 5.8: Resultados da implementação sequencial de GRASP com uso de *path-relinking* para as instâncias de ORLIB agrupadas por configurações.

			GRASP sequencial com PR				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	62	0,0000	0,1440	0,0464	0,9826
		0,50	140	0,0000	0,0461	0,0224	1,5664
		0,75	126	0,0000	0,0325	0,0097	1,5146
	250	0,25	7	0,0000	0,1305	0,0408	6,4417
		0,50	5	0,0000	0,0607	0,0189	6,3567
		0,75	11	0,0000	0,0331	0,0151	4,9821
	500	0,25	1	0,0000	0,1331	0,0286	17,3213
		0,50	2	0,0000	0,0496	0,0140	16,7791
		0,75	2	0,0000	0,0329	0,0089	13,5172
10	100	0,25	42	0,0000	0,4769	0,0992	1,3855
		0,50	60	0,0000	0,1920	0,0499	2,2367
		0,75	171	0,0000	0,0545	0,0277	2,1158
	250	0,25	1	0,0000	0,2962	0,0698	9,4640
		0,50	0	0,0376	0,1790	0,0296	9,0254
		0,75	22	0,0000	0,0757	0,0248	7,0644
	500	0,25	0	0,0696	0,2312	0,0401	23,9152
		0,50	0	0,0309	0,1422	0,0225	22,6806
		0,75	0	0,0112	0,0737	0,0153	18,6863
30	100	0,25	4	0,0000	0,9867	0,24887	2,9319
		0,50	0	0,0872	0,4829	0,0838	5,0581
		0,75	5	0,0000	0,2682	0,0319	4,1677
	250	0,25	0	0,2903	0,8225	0,1357	20,6638
		0,50	0	0,2536	0,4725	0,0609	20,7103
		0,75	0	0,0943	0,2351	0,0347	14,3478
	500	0,25	0	0,2773	0,7637	0,0709	50,4715
		0,50	0	0,1648	0,3669	0,0338	48,6467
		0,75	0	0,1121	0,2182	0,0243	39,1340

Tabela 5.9: Resultados da implementação sequencial de GRASP com uso de *path-relinking* para as instâncias de GK.

		GRASP sequencial com PR				
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
15	100	0	0,1859	0,2399	0,0325	2,1613
25	100	0	0,3790	0,6165	0,1179	3,6411
25	150	0	0,7249	0,8903	0,1184	8,1372
50	150	0	0,3641	0,4944	0,0720	14,2145
25	200	0	0,0397	0,1764	0,0497	15,1128
50	200	0	0,8603	1,1079	0,0947	23,5639
25	500	0	0,2238	0,2996	0,0391	90,9779
50	500	0	0,2393	0,2771	0,0220	162,1004
25	1500	0	0,8591	0,9320	0,0363	391,9932
50	1500	0	1,1083	1,2051	0,0494	680,9785
100	2500	0	0,1901	0,2235	0,0245	2652,3866

Tabela 5.10: Resultados da implementação usando GPGPU de GRASP com uso de *path-relinking* para as instâncias de SAC-94.

		GRASP GPGPU com PR					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	0	1,3497	2,6578	0,6232	0,8280	0,2407
PB	6	0	1,7298	4,9350	0,1656	0,9256	0,1804
PET	6	0	0,1754	3,3348	0,0174	0,5340	0,4131
SENTO	2	0	0,2981	0,9082	0,0000	3,3304	0,5206
WEING	8	0	0,0085	2,3660	0,0000	0,7094	0,9320
WEISH	30	150	0,0000	0,5907	0,0000	1,3675	0,5186

Tabela 5.11: Resultados da implementação usando GPGPU de GRASP com uso de *path-relinking* para as instâncias de ORLIB agrupadas por configurações.

			GRASP GPGPU com PR					
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	66	0,0000	0,1777	0,0636	1,8253	0,5382
		0,50	94	0,0000	0,0729	0,0245	1,5959	0,9816
		0,75	111	0,0000	0,0763	0,0125	1,3731	1,1037
	250	0,25	5	0,0000	0,1581	0,0440	5,4202	1,1861
		0,50	1	0,0000	0,0783	0,0210	4,9325	1,2876
		0,75	11	0,0000	0,0446	0,0183	4,6706	1,0653
	500	0,25	0	0,0295	0,1600	0,0319	12,8037	1,3538
		0,50	0	0,0009	0,0656	0,0168	13,3248	1,2599
		0,75	0	0,0101	0,0445	0,0103	13,6919	0,9880
10	100	0,25	33	0,0000	0,5354	0,1089	2,6525	0,5220
		0,50	55	0,0000	0,2737	0,0494	2,2330	1,0033
		0,75	98	0,0000	0,1271	0,0360	1,8050	1,1786
	250	0,25	0	0,0706	0,3811	0,0768	7,6596	1,2355
		0,50	0	0,0496	0,2543	0,0479	6,9093	1,3071
		0,75	1	0,0000	0,1129	0,0214	6,1171	1,1549
	500	0,25	0	0,0895	0,2631	0,0428	17,8185	1,3428
		0,50	0	0,0790	0,1694	0,0250	18,3407	1,2360
		0,75	0	0,0112	0,0942	0,0169	18,1007	1,0326
30	100	0,25	0	0,3525	1,0629	0,2554	5,5727	0,5275
		0,50	0	0,1583	0,6251	0,1198	4,7119	1,0736
		0,75	0	0,1036	0,3977	0,0611	3,4985	1,1916
	250	0,25	0	0,4003	1,1037	0,1390	16,4209	1,2586
		0,50	0	0,2580	0,5947	0,0642	15,0495	1,3757
		0,75	0	0,1458	0,3005	0,0404	12,5571	1,1430
	500	0,25	0	0,4095	0,9091	0,0878	39,0108	1,2933
		0,50	0	0,1977	0,4333	0,0358	40,1500	1,2119
		0,75	0	0,1568	0,2564	0,0245	37,9123	1,0320

Tabela 5.12: Resultados da implementação usando GPGPU de GRASP com uso de *path-relinking* para as instâncias de GK.

		GRASP GPGPU com PR					
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
15	100	0	0,1859	0,2452	0,0296	2,2934	0,9424
25	100	0	0,5306	0,8127	0,2115	3,2913	1,1063
25	150	0	1,0078	1,1434	0,1501	5,7151	1,4238
50	150	0	0,3988	0,7420	0,1236	9,4217	1,5087
25	200	0	0,2779	0,4014	0,0587	8,5730	1,7628
50	200	0	0,9906	1,2826	0,1254	14,7599	1,5965
25	500	0	0,2706	0,3577	0,0474	33,3846	2,7251
50	500	0	0,3669	0,3931	0,0297	54,3761	2,9811
25	1500	0	0,9813	1,0858	0,0657	223,2568	1,7558
50	1500	0	1,1886	1,3501	0,0742	419,0884	1,6249
100	2500	0	0,2279	0,2689	0,0173	1834,6083	1,4458

Os valores apresentados nas tabelas sugerem que a associação de *path-relinking* foi capaz de melhorar as soluções alcançadas pelo GRASP, embora acompanhado de um pequeno aumento no tempo de execução. Na implementação sequencial, o uso da intensificação levou a *gaps* menores em 26 das 27 configurações das instâncias de ORLIB e em todas as instâncias de GK; para as instâncias de SAC-94, os *gaps* foram menores ou iguais aos obtidos sem *path-relinking*. Entretanto, quando os resultados obtidos pelas implementações usando GPGPU são comparados aos obtidos pelas implementações sequenciais equivalentes, nota-se que houve um pequeno *speedup*, mas acompanhada de uma redução na qualidade dos resultados. Ainda assim, as soluções obtidas pela versão paralela foram competitivas, com *gaps* de no máximo 4,94, ocorrido nas instâncias de SAC-94. De fato, a heurística apresentou os resultados menos representativos nas instâncias relativamente pequenas de SAC-94. Presume-se que o número muito reduzido de itens e restrições das instâncias desse conjunto faz com que a busca realizada por GRASP não consiga atingir a variabilidade necessária para escapar dos máximos locais de baixa qualidade e, com isso, soluções melhores não são alcançadas.

Para demonstrar a viabilidade do uso de GRASP com *path-relinking* para solucionar o problema da mochila multidimensional, foi feito um comparativo das soluções alcançadas para as instâncias da ORLIB pelas implementações propostas com as obtidas pelo algoritmo genético de Chu e Beasley (AG-CB) [17], pela abordagem neurogenética de Deane e Agarwal [23], pelo algoritmo de otimização por colônia de formigas de Fingler *et al.* (OCF) [28], pelo método de programação dinâmica com melhoria nas computações de limite inferior (*hybrid dynamic programming method with an improvement in lower bound computations* – HDP + LBC) de Boyer, Elkihel e El Baz [11] e pelo novo método de redução de problemas de Hill, Cho e Moore (*new method problem reduction* – NP(R)) [35]. Os gráficos das Figuras 5.1 e 5.2 ilustram os *gaps* e os tempos de execução, atualizados pelo fator de correção descrito no Capítulo 1 dos algoritmos comparados, exceto os tempos do algoritmo genético e do NP(R), os quais são muito maiores que os demais.

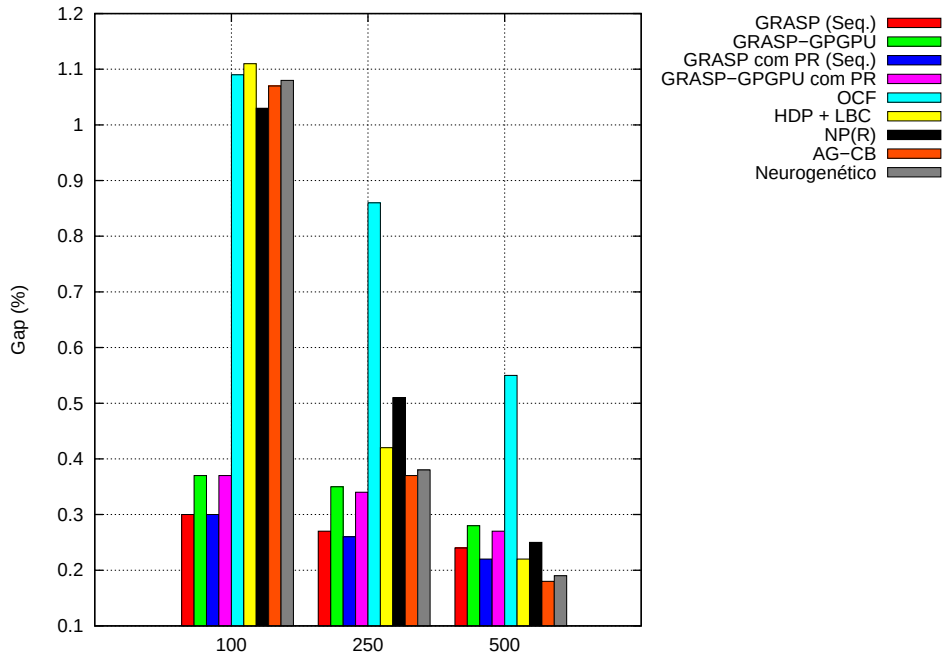


Figura 5.1: *Gaps* agrupados por número de itens.

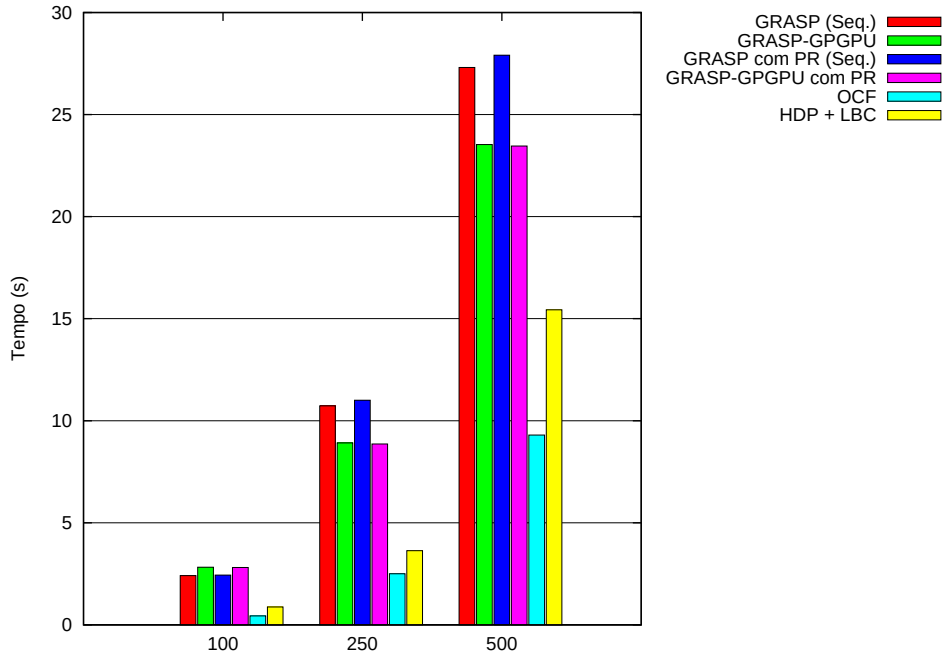


Figura 5.2: Tempos de execução agrupados por número de itens.

Pela análise do gráfico da Figura 5.1, pode-se notar que GRASP levou a *gaps* menores que os das outras abordagens, exceto para as instâncias com 500 itens para as quais os *gaps* resultantes da implementação sequencial foram discretamente maiores do que os alcançados pelo algoritmo genético e a abordagem neurogenética, os *gaps* da versão usando GPGPU também foram maiores do que os obtidos pelos algoritmos HDP + LBC e NP(R). Entretanto, as melhoras nos resultados alcançados nas instâncias com

100 e 250 itens são significativamente maiores que a redução de qualidade ocorrida nas instâncias com 500 itens.

Os resultados para as instâncias do conjunto GK foram comparadas aos obtidos pelas duas configurações da abordagem híbrida baseada no algoritmo de distribuição de estimativas de Wang, Wang e Xu (HEDA1 e HEDA2) [74], pela abordagem híbrida de programação linear e busca tabu de Vasquez e Hao (TS^{MKP}) [71], a abordagem heurística usando matriz de intercepção de Veni e Balachandar (DPHEU) [72]. A Tabela 5.13 mostra os *gaps* obtidos pelos algoritmos e pelo CPLEX (valores apresentados em [72]), usado como referencial. Os valores das melhores soluções obtidas são destacados em negrito.

Tabela 5.13: Comparação dos resultados para o conjunto GK.

m	n	PR-GRASP	GPGPU-PR-GRASP	HEDA1	HEDA2	TS^{MKP}	DPHEU	CPLEX
15	100	3756,97	3756,77	3754,05	3737,75	**3766**	**3766**	**3766**
25	100	3933,60	3925,83	3950,15	3919,25	**3958**	**3958**	**3958**
25	150	5599,70	5585,40	5638,60	5584,85	**5656**	**5656**	5650
50	150	5735,50	5721,23	5744,52	5690,95	**5767**	**5767**	5764
25	200	7543,67	7526,67	7536,05	7443,95	**7560**	7557	7557
50	200	7587	7573,60	7637,95	7561,25	**7677**	7672	7672
25	500	19157,43	19146,27	19076,59	18860,29	**19220**	19215	19215
50	500	18748,90	18727,10	18656,61	18476,49	**18806**	**18806**	18801
25	1500	57543,63	57454,33	–	56606,10	**58087**	58085	58085
50	1500	56601,57	56518,50	–	56079,13	**57295**	57294	57292
100	2500	95018,17	94974,93	–	93578,93	**95237**	95231	95231

Os tempos de execução não puderam ser comparados, pois os trabalhos de referência não os apresentam para as instâncias maiores presentes no conjunto GK.

A validação dos resultados foi realizada comparando os valores das soluções alcançadas por cada uma das implementações de GRASP com os valores de referência previamente descritos. Os p-valores calculados para as instâncias de SAC-94 e para as diferentes configurações de OR são apresentados nas Tabela 5.14, enquanto os valores de W obtidos para as instâncias de GK por cada implementação são ilustrados na Tabela 5.15

Tabela 5.14: p-valores calculados para SAC-94 e OR.

		p-valor			
m	n	Sequencial	GPGPU	Sequencial com PR	GPGPU com PR
SAC-94		0,3994	0,4029	0,4006	0,4053
5	100	0,4151	0,4094	0,4094	0,4094
	250	0,3951	0,3837	0,3951	0,3837
	500	0,3894	0,3894	0,3951	0,3951
10	100	0,3476	0,3260	0,3476	0,3395
	250	0,3395	0,3287	0,3449	0,3287
	500	0,3614	0,3614	0,3669	0,3614
30	100	0,3287	0,3076	0,3287	0,3076
	250	0,2081	0,1646	0,2123	0,1683
	500	0,2210	0,1836	0,2299	0,1997

Tabela 5.15: Valores de W calculados para as instâncias de GK.

	Sequencial	GPGPU	Sequencial com PR	GPGPU com PR
W	121	119	121	121

De acordo com os valores obtidos, pode-se concluir que as soluções obtidas para as instâncias de teste não são significativamente diferentes das soluções de referência.

5.4 Contribuições

O desenvolvimento das diferentes implementações de GRASP descritas neste capítulo levaram a contribuições importantes, com destaque a:

- Inclusão da intensificação por *path-relinking* no GRASP para o problema da mochila multidimensional;
- Publicação de artigo em *International Conference On Computational Science* (ICCS 2015) [20] com os resultados obtidos sem a utilização de *path-relinking*);
- Constatação de que, devido à possibilidade de exploração de várias regiões, GRASP adequou-se melhor a instâncias maiores, permitindo escapar com mais facilidade de máximos locais de baixa qualidade;
- Percepção de que a busca local implementada limita o desempenho da implementação paralela.

CAPÍTULO 6

Simulated Annealing

6.1 Fundamentação Teórica

Simulated annealing (SA) é uma metaheurística probabilística, proposta por Kirkpatrick, Gellet e Vecchi [41] e por Cerny [16], que se baseia nos princípios da termodinâmica para encontrar soluções para problemas de otimização. A técnica emula o processo físico de esfriar lentamente um sólido até um ponto de energia mínimo, no qual o sólido está congelado [9].

Os elementos básicos utilizados na técnica de *simulated annealing* são [9]:

- Um conjunto finito S;
- Uma função de custo real J definida em S;
- Para cada $i \in S$, tem-se um conjunto $S(i) \subset S \setminus \{i\}$ denominado o conjunto de vizinhos de i;
- Para cada i, considera-se uma coleção de coeficientes positivos $q_{ij}, j \in S(i)$ tais que $\sum_{j \in S(i)} q_{ij} = 1$. Assume-se que $j \in S(i)$ se, e somente se, $i \in S(j)$;
- Uma função não crescente $T : N \to (0, \infty)$, onde N é o conjunto dos números naturais e $T(t)$ é a temperatura no tempo t;
- Um estado inicial $x(0) \in S$.

Considerando tais elementos, o algoritmo de *simulated annealing* consiste em uma cadeia de Markov $x(t)$ não-homogênea. Considerando i como o estado atual $x(t)$, escolhe-se um vizinho aleatório $j \in S(i)$ com probabilidade igual a q_{ij}; o estado $x(t+1)$ é determinado baseado na seguinte análise para um problema de minimização: se $J(j) \leq J(i)$, então $x(t+1) = j$, caso contrário, $x(t+1) = j$ com probabilidade $\exp[-(J(j) - J(i))/T(t)]$ ou $x(t+1) = i$ [9]. Essa é a estratégia usada pelo algoritmo para tentar escapar de ótimos locais, ela permite que a busca pela solução faça movimentos para

soluções inferiores com uma certa probabilidade, que é gradualmente reduzida à medida que a temperatura diminui.

De acordo com Ferreiro *et al.* [25], o algoritmo *simulated annealing* é inerentemente sequencial o que torna difícil a sua paralelização sem afetar a sua natureza recursiva, apesar disso diversas estratégias foram propostas para a sua paralelização, incluindo [25]:

- Paralelização dependente de aplicação: as operações da função de custo são decompostas entre os processadores;
- Decomposição de domínio: o espaço de busca é dividido em vários subdomínios, cada processador procura o mínimo (máximo) em seu subdomínio local e compartilha o resultado com os demais processadores;
- Múltiplas cadeias de Markov: múltiplas cadeias de Markov são executadas de maneira independente pelos processadores, os quais comunicam seus estados somente no final do processo ou periodicamente em espaços de tempo determinados. A primeira abordagem, mais direta, é denominada *assíncrona* e a segunda *síncrona*. Em implementações síncronas, surgem questões a respeito da periodicidade dos envios dos estados e de como eles serão implementados; quanto maior o número de trocas, maior é a sobrecarga causada pela comunicação, entretanto, com a redução desse número, o comportamento tende ao da abordagem assíncrona.

Onbaşoğlu e Özdamar [55] também propuseram cinco diferentes versões de SA paralelo e compararam os resultados obtidos com uma extensa base de comparações previamente utilizada para classificar a qualidade de soluções para problemas de otimização combinatória. Os algoritmos propostos pelos autores podem ser classificados em diferentes categorias: abordagem assíncrona e abordagens síncronas nas quais soluções são trocadas entre os elementos de processamento usando operadores genéticos ou transmitidas ocasionalmente ou, ainda, a cada iteração. Zbierski [77] propôs um algoritmo SA paralelo, denominado *Simulated Annealing de Dois Níveis* (*Two-Level Simulated Annealing* – TLSA), otimizado para execução em *clusters* de GPU e que se aproveita das características providas por tais ambientes através da decomposição hierárquica do problema a ser resolvido.

6.2 Implementação

O algoritmo de *simulated annealing* foi implementado em uma versão sequencial e em uma versão paralela usando GPGPU, ambas baseadas nas ideias apresentadas por Qian e Ding [59] considerando implementações com tempo delimitado, que não garantem que a solução ótima será encontrada, mas permitem tempos de execução menores sem grandes perdas na qualidade das soluções. O Algoritmo 6.1 ilustra os passos do programa sequencial implementado e recebe como entrada as temperaturas inicial (t_0) e final (t_f), o tamanho da cadeia de Markov (tam) que define o número de iterações executadas a cada temperatura, um fator β que controla a taxa de diminuição da temperatura e uma constante α utilizada para a construção da solução inicial.

Algoritmo 6.1: SimulatedAnnealing(t_0, t_f, *tam*, β, α)

```
1  LêEntrada();
2  Semente ← um valor baseado no tempo;
3  Solução ← ConstróiSoluçãoInicial(Semente, alpha);
4  MelhorSolução ← Solução;
5  temp ← t_0;
6  enquanto temp ≥ t_f faça
7      para it ← 0, tam faça
8          NovaSolução ← Solução;
9          Escolhe um índice aleatório i ∈ {1,...,n};
10         se item i ∉ Solução então
11             Adiciona item i a NovaSolução;
12             enquanto NovaSolução não é viável faça
13                 Remove item aleatório de NovaSolução;
14             fim
15             Δ ← f(NovaSolução) - f(Solução);
16             num ← valor aleatório entre 0 e 1;
17             se Δ ≥ 0 ou num < exp[Δ/temp] então
18                 Solução ← NovaSolução;
19             fim
20         senão
21             Remove item i de NovaSolução;
22             Adiciona um novo item aleatório a NovaSolução;
23             Δ ← f(NovaSolução) − f(Solução);
24             num ← valor aleatório entre 0 e 1;
25             se Δ ≥ 0 ou num < exp[Δ/temp] então
26                 Solução ← NovaSolução;
27             fim
28         fim
29         se f(Solução) > f(MelhorSolução) então
30             MelhorSolução ← Solução;
31         fim
32     fim
33     temp ← β * temp;
34 fim
35 retorna MelhorSolução;
```

Em uma tentativa de escapar das abordagens aleatórias tradicionais, nas implementações realizadas utilizou-se a mesma estratégia para construção da solução inicial utilizada na metaheurística GRASP. Para tal, ao invés de adicionar aleatoriamente itens à solução até que nenhum outro item possa ser adicionado, constrói-se uma RCL e os itens são adicionados aleatoriamente a partir dessa lista. O uso dessa estratégia possibilitou que a implementação proposta obtivesse soluções iniciais melhores do que as obtidas por um método puramente aleatório, consequentemente, as soluções finais alcançadas tiveram melhor qualidade do que as obtidas por Qian e Ding.

A versão GPGPU foi implementada usando a abordagem assíncrona de múltiplas cadeias de Markov, descrita em [25]. Essa estratégia foi escolhida por possibilitar uma

adaptação mais intuitiva a partir do algoritmo sequencial e, apesar de sua relativa simplicidade, tornou possível alcançar resultados satisfatórios. Nessa implementação, a CPU é responsável pela leitura dos dados de entrada, pela determinação do tamanho da cadeia de Markov a ser executada por cada *thread* e por invocar as funções de kernel para configurar a semente do gerador de números aleatórios (`ConfiguraKernel()`), construir a solução inicial (`ConstróiSoluçãoInicial()`) e executar o laço de variação de temperatura (`PrincipalKernel()`). As t *threads* da GPU executam em paralelo cada uma dessas funções de kernel e o tamanho da cadeia de Markov executada por cada *thread* é igual a tam/t, onde tam é tamanho da cadeia de Markov fornecido como entrada do programa. Para possibilitar que as *threads* explorem vizinhanças diferentes, a função `ConfiguraKernel()` configura o gerador de números aleatórios de cada *thread* com diferentes sementes. Os Algoritmos 6.2 e 6.3 ilustram, respectivamente, os passos do *simulated annealing* usando GPGPU e do *kernel* responsável por executar a cadeia de Markov nas *threads*.

Algoritmo 6.2: SimulatedAnnealing_GPGPU(t_0, t_f, tam, β, α, $numBlocos$, $numThreads$)

```
1 LêEntrada();
2 ConfiguraKernel();
3 ConstróiSoluçãoInicialKernel(α);
4 tam ← tam/(numBlocos * numThreads);
5 PrincipalKernel(ThreadMelhorSolução, tam, β);
6 MelhorSolução ← melhor solução encontrada pelas threads;
7 retorna MelhorSolução;
```

Algoritmo 6.3: PrincipalKernel(*ThreadMelhorSolução*, *tam*, β)

```
1  Solução ← ThreadMelhorSolução;
2  enquanto temp ≥ t_f faça
3      para it ← 0, tam faça
4          NovaSolução ← Solução;
5          Seleciona um item aleatório i ∈ {1,...,n};
6          se item i ∉ Solução então
7              Adiciona item i a NovaSolução;
8              enquanto NovaSolução não é viável faça
9                  Remove item aleatório de NovaSolução;
10             fim
11             Δ ← f(NovaSolução) − f(Solução);
12             num ← valor aleatório entre 0 e 1;
13             se Δ ≥ 0 ou num < exp[Δ/temp] então
14                 Solução ← NovaSolução;
15             fim
16         senão
17             Remove item i de NovaSolução;
18             Adiciona um novo item aleatório a NovaSolução;
19             Δ ← f(NovaSolução) − f(Solução);
20             num ← valor aleatório entre 0 e 1;
21             se Δ ≥ 0 ou num < exp[Δ/temp] então
22                 Solução ← NovaSolução;
23             fim
24         fim
25         se f(Solução) > f(ThreadMelhorSolução) então
26             ThreadMelhorSolução ← Solução;
27         fim
28     fim
29     temp ← β * temp;
30 fim
31 retorna ThreadMelhorSolução;
```

6.3 Resultados

Os algoritmos desenvolvidos foram inicialmente executados com basicamente as mesmas configurações usadas em [59]:

- Temperatura inicial: 500;
- Temperatura final: 0,00001;
- Tamanho da cadeia de Markov: igual ao número de itens, exceto nas instâncias do conjunto SAC-94, para as quais foi fixado o tamanho 100, pois, dado que a implementação em GPGPU divide o tamanho da cadeia pelas *threads* participantes, para instâncias com número de itens muito pequeno a qualidade da solução seria prejudicada;

- Taxa de atualização da temperatura: 0,85.

A observação dos resultados obtidos com essa configuração mostrou que a implementação sequencial obteve resultados de qualidade inferior aos alcançados pela versão paralela, entretanto, tais resultados também demandaram menor tempo de execução. Com base nessa situação, experimentou-se aumentar em dez vezes o tamanho da cadeia de Markov na versão sequencial para verificar o impacto do aumento do número de iterações na qualidade das soluções construídas. Os novos resultados mostraram que, mesmo assim, os *gaps* obtidos pela versão sequencial continuam maiores que os alcançados pela implementação GPGPU, o que indica que a variabilidade proporcionada pela execução em múltiplas *threads* de uma GPU e, consequentemente, a exploração de um número maior de vizinhanças, é mais eficaz na busca por soluções de melhor qualidade do que explorar exaustivamente uma vizinhança única. Os resultados obtidos pela implementação sequencial com a cadeia de Markov de tamanho igual a dez vezes o número de itens de cada instância são apresentados nas Tabelas 6.1, 6.2 e 6.3.

Tabela 6.1: Resultados da implementação sequencial de *simulated annealing* para as instâncias de SAC-94.

		SA Sequencial				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	8	0,0000	1,2173	0,6675	0,1328
PB	6	23	0,0000	2,6534	1,4243	0,1528
PET	6	46	0,0000	0,4818	0,2600	0,1013
SENTO	2	28	0,0000	0,1055	0,1698	0,3811
WEING	8	121	0,0000	0,6949	0,2311	0,1475
WEISH	30	443	0,0000	0,2570	0,2003	0,2401

Tabela 6.2: Resultados da implementação sequencial de *simulated annealing* para as instâncias de ORLIB agrupadas por configurações.

			SA Sequencial				
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	0	0,1334	1,0186	0,4545	0,5497
		0,50	0	0,0690	0,5491	0,2841	0,5787
		0,75	6	0,0000	0,2836	0,1352	0,6211
	250	0,25	0	0,1448	0,6715	0,2707	3,7789
		0,50	0	0,0448	0,2827	0,1212	4,8780
		0,75	0	0,0100	0,1603	0,0767	5,2721
	500	0,25	0	0,0784	0,3872	0,1539	21,7163
		0,50	0	0,0330	0,1675	0,0672	23,1644
		0,75	0	0,0304	0,1084	0,0431	25,7230
10	100	0,25	0	0,5395	1,9365	0,5908	0,5508
		0,50	0	0,2043	0,8454	0,2912	0,8654
		0,75	20	0,0000	0,4309	0,1901	0,8400
	250	0,25	0	0,2426	1,0437	0,3456	6,2051
		0,50	0	0,0762	0,5493	0,1675	7,5421
		0,75	0	0,0301	0,2733	0,0919	7,4512
	500	0,25	0	0,1553	0,5728	0,1881	33,7927
		0,50	0	0,1093	0,2943	0,0902	43,5361
		0,75	0	0,0403	0,1685	0,0535	42,4107
30	100	0,25	0	0,6674	2,3322	0,6363	0,5692
		0,50	0	0,1681	1,2608	0,3371	0,8382
		0,75	3	0,0000	0,7263	0,1773	0,8828
	250	0,25	0	0,9104	1,8452	0,3722	6,3510
		0,50	0	0,3891	0,8223	0,1772	8,6407
		0,75	0	0,2113	0,4609	0,0958	7,9770
	500	0,25	0	0,4476	1,2901	0,2331	43,3288
		0,50	0	0,2815	0,6103	0,1034	57,7179
		0,75	0	0,1783	0,3299	0,0556	52,1078

Tabela 6.3: Resultados da implementação sequencial de *simulated annealing* para as instâncias de GK.

		SA Sequencial				
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
15	100	0	0,2390	0,4983	0,1473	0,4109
25	100	0	0,4295	0,9197	0,2856	0,3516
25	150	0	0,5658	0,9512	0,1965	1,1744
50	150	0	0,5722	1,0653	0,2658	1,0209
25	200	0	0,3838	0,6180	0,1600	1,9129
50	200	0	0,6778	1,0658	0,2140	1,7390
25	500	0	0,2446	0,4602	0,1247	18,6645
50	500	0	0,3510	0,5909	0,1228	18,7049
25	1500	0	0,1463	0,2293	0,0483	384,7354
50	1500	0	0,2356	0,4543	0,0858	360,3986
100	2500	0	0,4232	0,5719	0,0909	1493,3889

As Tabelas 6.4, 6.5 e 6.6 mostram os valores obtidos com a implementação usando GPGPU do algoritmo de *simulated annealing* e os *speedups* alcançados com relação à versão sequencial.

Tabela 6.4: Resultados da implementação usando GPGPU de *simulated annealing* para as instâncias de SAC-94.

		SA GPGPU					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	30	0,0000	0,3374	0,0878	0,0241	5,4860
PB	6	77	0,0000	0,3525	0,0342	0,0312	5,3261
PET	6	120	0,0000	0,1246	0,0180	0,0201	5,0004
SENTO	2	30	0,0000	0,0314	0,0240	0,0927	4,2485
WEING	8	150	0,0000	0,4984	0,0189	0,0265	5,8857
WEISH	30	533	0,0000	0,1268	0,0221	0,0532	4,9549

Tabela 6.5: Resultados da implementação usando GPGPU de *simulated annealing* para as instâncias de ORLIB agrupados por configurações.

			SA GPGPU					
m	n	α	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	11	0,0000	0,4438	0,1024	0,1239	4,5044
		0,50	8	0,0000	0,2068	0,0391	0,1231	4,7035
		0,75	29	0,0000	0,1382	0,0384	0,1273	4,8699
	250	0,25	0	0,1383	0,2811	0,0670	0,7626	4,9894
		0,50	0	0,0448	0,1188	0,0259	0,9582	5,1153
		0,75	0	0,0100	0,0572	0,0160	0,9695	5,4449
	500	0,25	0	0,0319	0,1456	0,0335	6,2477	3,4964
		0,50	0	0,0009	0,0657	0,0162	6,3171	3,6845
		0,75	0	0,0101	0,0399	0,0091	6,9883	3,6747
10	100	0,25	0	0,5395	1,2690	0,1528	0,1120	5,0948
		0,50	0	0,1882	0,4957	0,0644	0,1599	5,4706
		0,75	53	0,0000	0,2306	0,0306	0,1626	5,1957
	250	0,25	0	0,1836	0,5953	0,0925	1,0014	6,2967
		0,50	0	0,0793	0,3456	0,0509	1,1084	6,9871
		0,75	0	0,0655	0,1545	0,0276	1,1834	6,3349
	500	0,25	0	0,0843	0,2646	0,0448	8,2386	4,1549
		0,50	0	0,0695	0,1552	0,0231	9,5844	4,5513
		0,75	0	0,0295	0,0845	0,0134	9,5484	4,4674
30	100	0,25	0	0,6674	1,8290	0,1872	0,1353	4,3108
		0,50	0	0,2630	0,9761	0,1786	0,1835	4,6014
		0,75	18	0,0000	0,6716	0,1175	0,1978	4,4523
	250	0,25	0	0,7946	1,5831	0,1454	0,9327	6,9481
		0,50	0	0,4547	0,7847	0,0625	1,2732	6,9214
		0,75	0	0,1227	0,3852	0,0468	1,4264	5,5982
	500	0,25	0	0,4501	1,0441	0,0755	7,5029	6,1177
		0,50	0	0,2616	0,5199	0,0354	9,4719	6,2029
		0,75	0	0,1488	0,2846	0,0219	9,5367	5,5183

Tabela 6.6: Resultados da implementação usando GPGPU de *simulated annealing* para as instâncias de GK.

		SA GPGPU					
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
15	100	0	0,2921	0,3514	0,0448	0,1112	3,6951
25	100	0	0,4800	0,6864	0,1435	0,1186	2,9646
25	150	0	0,6188	0,7402	0,0708	0,2925	4,0150
50	150	0	0,7803	0,9352	0,0677	0,3974	2,5689
25	200	0	0,3573	0,4358	0,0404	0,5820	3,2868
50	200	0	0,5605	0,7899	0,0810	0,7860	2,2125
25	500	0	0,1978	0,2581	0,0255	5,7002	3,2744
50	500	0	0,4148	0,4559	0,0198	6,9119	2,7062
25	1500	0	0,0895	0,1184	0,0114	117,4823	3,2748
50	1500	0	0,2548	0,3046	0,0178	103,6961	3,4664
100	2500	0	0,3560	0,4263	0,0238	380,6320	3,9234

Os valores apresentados nas tabelas mostram que o uso de GPGPU, mesmo com o menor número de iterações, tornou possível alcançar resultados melhores e em menor tempo do que a versão sequencial equivalente em todas as configurações de testes. Para mostrar que a implementação proposta de *simulated annealing*, é competitiva com outras abordagens heurísticas, ilustra-se na Tabela 6.7 a comparação dos *gaps* obtidos para as instâncias de ORLIB pela implementação GPGPU com os obtidos pela implementação sequencial de Qian e Ding (SA-QD) [59], pela otimização por colônia de formigas de Fingler *et al.* (OCF) [28], pelo algoritmo genético de Chu e Beasley (AG-CB) [17], pelo método híbrido de programação dinâmica com aperfeiçoamento nas computações de limites inferiores (HDP + LBC) proposto por Boyer, Elkihel e El Baz [11] e pelo novo método de redução de problemas (NP(R)) proposto por Hill, Cho e Moore [35].

Tabela 6.7: Resultados obtidos comparados aos de outros trabalhos de referência.

n	m	SA GPGPU	SA-QD	OCF	AG-CB	HDP+LBC	NP(R)
5	100	0,26	1,83	0,26	0,59	0,57	0,53
	250	0,15	1,94	0,24	0,14	0,16	0,24
	500	0,08	2,13	0,15	0,05	0,07	0,08
10	100	0,67	2,36	1,01	0,94	0,95	1,10
	250	0,37	2,21	0,63	0,30	0,32	0,48
	500	0,17	2,45	0,33	0,14	0,16	0,19
30	100	1,16	2,95	2,01	1,69	1,81	1,45
	250	0,92	2,27	1,70	0,68	0,77	0,80
	500	0,62	2,37	1,16	0,35	0,42	0,49

Pela análise dos resultados, percebe-se que a abordagem proposta supera a implementação de Qian e Ding e a da OCF em todas as configurações das instâncias da biblioteca OR. Os *gaps* calculados também se mostraram promissores na comparação às outras metaheurísticas cujos *gaps* foram maiores em diversas instâncias. Para possibilitar a comparação dos tempos de execução, apresentada no gráfico da Figura 6.1,

foi utilizado o fator de correção descrito no Capítulo 2. Os tempos médios dos algoritmos foram agrupados por número de itens, com exceção os de NP(R), cujos dados não estavam disponíveis no trabalho de referência, e os do AG-CB, cujos tempos de execução foram aproximadamente 100 vezes maiores que os demais.

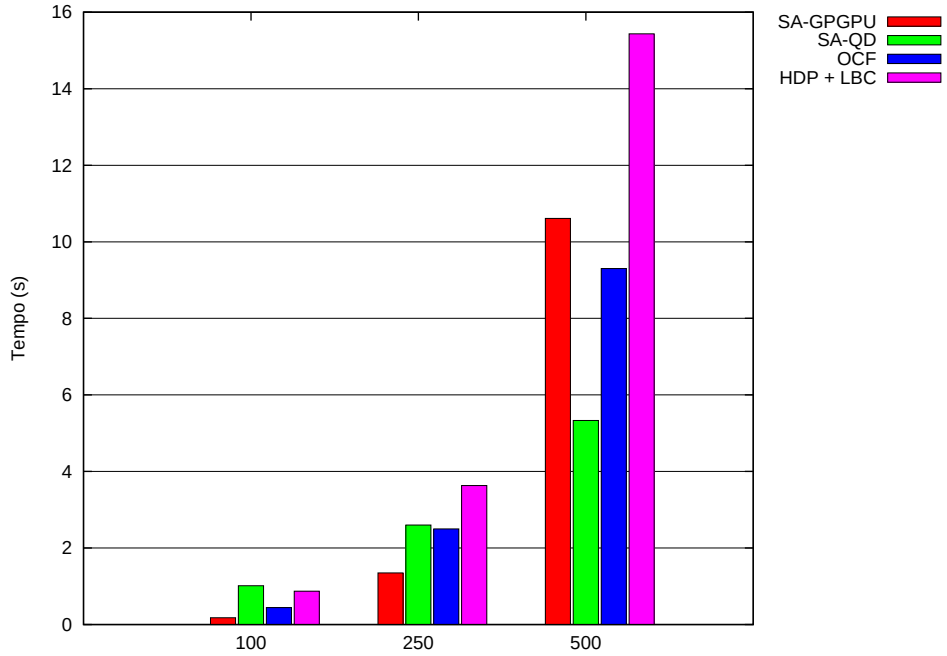


Figura 6.1: Tempos de execução agrupados por número de itens.

Conforme pode-se notar pelo gráfico, a implementação GPGPU proposta mostrou-se competitiva com as demais; ela executou mais rapidamente que os outros algoritmos nas instâncias com 100 e 250 itens, sendo mais lenta apenas que a OCF e a versão de SA de Qian e Ding nas instâncias com 500 itens.

As soluções obtidas pela implementações propostas de *simulated annealing* foram validadas usando o teste de somas de *ranks* de Wilcoxon. Para as instâncias de SAC-94, o p-valor calculado foi 0,4293 para o algoritmo sequencial e 0,4608 para o algoritmo GPGPU; os p-valores calculados para cada configuração de testes de ORLIB são mostrados na Tabela 6.8.

Tabela 6.8: p-valores calculados para cada configuração de testes de ORLIB.

		p-valor	
m	n	Sequencial	GPGPU
5	100	0,3234	0,3809
	250	0,3341	0,3781
	500	0,3449	0,3894
10	100	0,2344	0,2796
	250	0,2299	0,3024
	500	0,2713	0,3614
30	100	0,1720	0,2210
	250	0,1127	0,1275
	500	0,1469	0,1646

Na validação dos resultados para as instâncias de GK, em ambas as implementações, foram obtidos os valores de W iguais a 121, ou seja $W > W_{crit}$. Os p-valores e o valor de W calculados implicam que a hipótese nula do teste não pode ser negada, logo as soluções alcançadas não são significativamente diferentes das soluções de referência.

6.4 Contribuições

As implementações de *simulated annealing* apresentadas neste capítulo resultaram em contribuições relevantes, entre elas:

- Desenvolvimento de uma implementação paralela de SA para o problema da mochila multidimensional;
- Aceitação de artigo para publicação em *28th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2016);
- Proposição de nova alternativa de construção da solução inicial usando RCL;
- Exploração eficiente da cadeia de Markov em paralelo.

CAPÍTULO 7

Redes Neurais Aumentadas e GRASP

7.1 Fundamentação

Este capítulo descreve uma nova abordagem híbrida de redes neurais aumentadas e de GRASP, visando aumentar o espaço de soluções exploradas e incrementar a qualidade dos resultados obtidos. A proposta se baseou nas redes neurais aumentadas usando a heurística KMW, pois ela levou aos melhores resultados na implementação original. A ideia utilizada consistiu em alterar o método de escolha do item a ser adicionado a cada iteração de uma época: ao invés de sempre escolher o item com a maior pseudoutilidade, cria-se uma RCL com alguns dos melhores elementos e, então, escolhe-se um item aleatório dessa lista para ser adicionado à mochila. Essa proposta tenta controlar o quão gulosa é a escolha pelo próximo item e, assim, fazer com que seja mais fácil escapar de máximos locais de baixa qualidade.

7.2 Implementação

A implementação da rede neural aumentada híbrida foi realizada de maneira bastante semelhante à versão original descrita no Capítulo 4, acrescida do código para a construção da RCL e da escolha aleatória de um de seus elementos para adição à mochila. O tamanho da RCL é limitado de forma qualitativa de acordo com um fator α: somente os elementos e_i cuja pseudoutilidade $psUt_i$ estiver no intervalo $[psUt_{max} - \alpha * (psUt_{max} - psUt_{min}), psUt_{max}]$ são adicionados à RCL, onde $psUt_{max}$ e $psUt_{min}$ são as pseudoutilidades do melhor e do pior elemento, respectivamente. O Algoritmo 7.1 mostra os passos da implementação sequencial da abordagem híbrida de redes neurais aumentadas e GRASP.

Algoritmo 7.1: RNA_GRASP(*arq*, *taxa*, *max*, α)

```
1  LêEntrada();
2  enquanto sinal de entrada for diferente de zero faça
3  |  se sinal for igual a um então
4  |  |  Passa para a próxima iteração da mesma época;
5  |  senão
6  |  |  Remove todos itens da mochila;
7  |  |  Calcula erro da época anterior e atualiza vetor de pesos;
8  |  |  Incrementa contador de épocas e inicia nova época;
9  |  fim
10 |  Calcula pseudoutilidades dos nós ativos;
11 |  Constrói RCL;
12 |  Escolhe aleatoriamente um item viável s da RCL;
13 |  se s foi encontrado então
14 |  |  Adiciona s à solução;
15 |  |  Envia sinal 1 à camada de entrada;
16 |  senão
17 |  |  se número de épocas for igual a max então
18 |  |  |  Envia sinal 0 à camada de entrada;
19 |  |  senão
20 |  |  |  Envia sinal 2 à camada de entrada;
21 |  |  fim
22 |  fim
23 fim
```

A versão paralela da nova abordagem proposta também foi implementada usando GPGPU. A CPU é responsável pela leitura do arquivo de entrada e por alocar memória para o armazenamento das estruturas necessárias. Os dados lidos são, então, transferidos para a memória da GPU e as *threads* executam passos equivalentes aos do programa sequencial, cada uma construindo separadamente uma solução viável para o problema. A CPU também é responsável por gerenciar as épocas, assegurando que cada *thread* execute $max/(numBlocos * numThreads)$ épocas, onde max é o número de épocas fornecido como entrada, $numBlocos$ e $numThreads$ são, respectivamente, o número de blocos e de *threads* por bloco do *grid* configurado para execução na GPU. A cada época, a CPU invoca a função de kernel `ConfiguraÉpocaKernel()` que realiza operações para iniciar a construção de uma solução: remove os itens da mochila, restaura o vetor de capacidades remanescentes para o valor das capacidades de cada recurso e atualiza os pesos dos itens de acordo com o erro da época anterior (ou atribui o valor 1 a cada um deles, no caso da primeira época). A função `MeuKernel()`, por sua vez, implementa as atividades de uma única época da RNA híbrida; a cada vez que essa função é invocada, cada uma das $numBlocos*numThreads$ *threads* constrói uma solução viável própria e calcula o valor do erro a ela associado. Depois do retorno da chamada a `MeuKernel()`, a CPU encontra e armazena a melhor solução dentre as encontradas pelas *threads*. Os Algoritmos 7.2, 7.3 e 7.4 mostram os passos seguidos pela implementação GPGPU proposta, onde $fator$ é o

parâmetro que limita o tamanho da RCL e α é a taxa de aprendizado da RNA.

Algoritmo 7.2: GPGPU_RNA_GRASP($fator$, max, $numBlocos$, $numThreads$, α)

```
1 LêEntrada();
2 época ← 1;
3 enquanto época ≤ max/(numBlocos * numThreads) faça
4 |   ConfiguraÉpocaKernel(Solução, RC, W);
5 |   MeuKernel(Solução, RC, W);
6 |   CPU encontra e armazena melhor solução encontrada pelas threads;
7 |   época ← época +1;
8 fim
```

Algoritmo 7.3: ConfiguraÉpocaKernel(*Solução*, RC, W)

```
1 Solução ← ∅;
2 Restaura o vetor de capacidades remanescentes RC;
3 Atualiza o vetor de pesos W;
```

Algoritmo 7.4: MeuKernel(*Solução*, RC, W)

```
1 repita
2 |   Calcula pseudoutilidades dos itens ativos;
3 |   Constrói RCL;
4 |   Seleciona aleatoriamente um item viável s da RCL;
5 |   se s foi encontrado então
6 |   |   Solução ← Solução ∪ {s};
7 |   |   Atualiza vetor de capacidades remanescentes RC;
8 |   fim
9 até que nenhum item possa ser adicionado a Solução;
10 Calcula o valor do gap de Solução e atualiza o vetor de pesos W;
```

7.3 Resultados

Os testes realizados com as implementações híbridas usaram as mesmas configurações das redes neurais aumentadas puras, descritas no Capítulo 4. Foram testados dois valores para o fator limitante da RCL: 0,02 e 0,1. A primeira possibilidade, mais gulosa, levou a melhores resultados nas instâncias maiores de GK, enquanto a segunda levou a soluções de melhor qualidade nas instâncias de SAC-94. Presume-se que essa situação ocorra porque com o uso de valores maiores do fator limitante, a exploração do espaço de soluções se torna errática, com um número muito grande de itens passíveis de adição a cada iteração da rede. No caso das instâncias de ORLIB, o padrão permanece, as instâncias menores, com 100 itens, são favorecidas pelo uso de um fator limitante maior; enquanto as com 250 e 500 itens tiveram melhores resultados com o uso do fator igual a 0,02. As Tabelas 7.1, 7.2 e 7.3 mostram os resultados médios obtidos pela implementação sequencial considerando 10000 épocas; para as instâncias de SAC-94 e de ORLIB com 100 itens foi usado o fator limitante 0,1 e para as de ORLIB com 250 e 500 itens e para as de GK foi usado o fator 0,02.

Tabela 7.1: Resultados da implementação sequencial do híbrido de RNA e GRASP para as instâncias de SAC-94.

		RNA+GRASP Sequencial				
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
HP	2	10	0,0000	0,5188	0,2406	0,3425
PB	6	74	0,0000	0,3631	0,3269	0,4120
PET	6	100	0,0000	0,1272	0,0497	0,3775
SENTO	2	26	0,0000	0,0337	0,0555	2,5006
WEING	8	78	0,0000	0,1463	0,0467	0,6312
WEISH	30	834	0,0000	0,0080	0,0115	0,9230

Tabela 7.2: Resultados da implementação sequencial do híbrido de RNA e GRASP para as instâncias de ORLIB agrupadas por configurações.

			RNA+GRASP Sequencial				
m	n	δ	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
5	100	0,25	2	0,0000	0,5855	0,2295	1,4757
		0,50	3	0,0000	0,3079	0,1332	2,7261
		0,75	32	0,0000	0,1733	0,0598	3,9343
	250	0,25	0	0,0521	0,4651	0,1460	9,5765
		0,50	0	0,0448	0,1973	0,0637	17,9854
		0,75	1	0,0000	0,1116	0,0378	26,1574
	500	0,25	0	0,0697	0,2201	0,0533	38,3647
		0,50	0	0,0116	0,0958	0,0264	73,0664
		0,75	0	0,0197	0,0582	0,0158	106,4533
10	100	0,25	4	0,0000	1,2985	0,3568	1,9485
		0,50	1	0,0000	0,7121	0,1800	3,8101
		0,75	10	0,0000	0,3094	0,1116	5,5857
	250	0,25	0	0,2281	0,7926	0,1746	13,0976
		0,50	0	0,0793	0,4430	0,0931	25,6046
		0,75	0	0,0372	0,2324	0,0531	37,7329
	500	0,25	0	0,1135	0,3746	0,0878	52,4496
		0,50	0	0,0422	0,2038	0,0377	103,3474
		0,75	0	0,0349	0,1149	0,0240	153,1733
30	100	0,25	0	0,3098	2,0043	0,4352	3,9209
		0,50	0	0,4190	1,1178	0,2329	8,0745
		0,75	1	0,0000	0,5466	0,1474	12,3124
	250	0,25	0	0,8476	1,9728	0,2970	27,4311
		0,50	0	0,4287	0,9314	0,1218	55,4773
		0,75	0	0,2014	0,4489	0,0683	83,8505
	500	0,25	0	0,6195	1,2978	0,1621	112,1446
		0,50	0	0,3107	0,5958	0,0622	225,6910
		0,75	0	0,2075	0,3344	0,0365	338,9649

Tabela 7.3: Resultados da implementação sequencial do híbrido de RNA e GRASP para as instâncias de GK.

		RNA+GRASP Sequencial				
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)
15	100	0	0,4249	0,7603	0,1501	5,3671
25	100	0	0,7074	1,2978	0,3431	7,8410
25	150	0	0,9371	1,4244	0,2414	17,5549
50	150	0	0,9190	1,3300	0,2279	31,4080
25	200	0	0,5425	0,8509	0,1389	31,1515
50	200	0	0,9124	1,4347	0,2138	55,6047
25	500	0	0,3331	0,4358	0,0585	195,9820
50	500	0	0,4733	0,7260	0,1308	349,9812
25	1500	0	0,1567	0,2019	0,0272	1772,7040
50	1500	0	0,4102	0,5105	0,0497	3178,0086
100	2500	0	0,3255	0,4932	0,1299	16673,4771

Para a execução da versão paralela do híbrido foram usadas basicamente as mesmas configurações dos testes da rede neural aumentada paralela descrita no Capítulo 4, ou seja, 1000 épocas usando um *grid* com 25 blocos de 4 *threads* cada. O fator limitante da RCL, da mesma forma que na versão sequencial, foi testado com os valores 0,02 e 0,1. Assim como ocorreu na RNA, mesmo com a execução de um número menor de épocas, a implementação paralela levou a resultados de melhor qualidade e alcançou *speedups* significativos. Os resultados médios obtidos para os conjuntos de testes são apresentados nas Tabelas 7.4, 7.5 e 7.6; a configuração do fator limitante foi igual à versão sequencial: 0,1 para as instâncias de SAC-94 e para as de ORLIB com 100 itens, 0,02 para as demais instâncias de ORLIB e para as de GK.

Tabela 7.4: Resultados da implementação usando GPGPU do híbrido de RNA e GRASP para as instâncias de SAC-94.

		RNA+GRASP GPGPU					
Conjunto	Número de Problemas	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
HP	2	0	0,4681	1,7234	0,3925	0,0507	6,7544
PB	6	1	0,0000	2,3967	0,7080	0,0550	7,4886
PET	6	37	0,0000	0,6414	0,0671	0,0444	8,5090
SENTO	2	0	0,2981	0,9082	0,0000	0,2450	10,2086
WEING	8	0	0,0085	0,8782	0,2107	0,0727	8,6877
WEISH	30	213	0,0000	0,5611	0,0272	0,0971	9,5078

Tabela 7.5: Resultados da implementação usando GPGPU do híbrido de RNA e GRASP para as instâncias de ORLIB agrupadas por configurações.

			RNA+GRASP GPGPU					
m	n	δ	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
5	100	0,25	79	0,0000	0,2143	0,0993	0,1626	9,0741
		0,50	45	0,0000	0,1044	0,0487	0,2609	10,4496
		0,75	76	0,0000	0,0857	0,0280	0,3403	11,5615
	250	0,25	0	0,0358	0,2134	0,0453	1,0627	9,0113
		0,50	4	0,0000	0,1235	0,0235	1,7188	10,4641
		0,75	2	0,0000	0,0607	0,0115	2,2503	11,6242
	500	0,25	0	0,0340	0,1069	0,0267	4,0973	9,3633
		0,50	0	0,0055	0,0516	0,0119	6,6636	10,9650
		0,75	0	0,0036	0,0299	0,0071	8,7045	12,2297
10	100	0,25	18	0,0000	0,8377	0,1722	0,2093	9,3104
		0,50	28	0,0000	0,3687	0,1063	0,3555	10,7188
		0,75	62	0,0000	0,1516	0,0377	0,4621	12,0889
	250	0,25	0	0,2127	0,4695	0,0780	1,3941	9,3948
		0,50	0	0,0793	0,2917	0,0421	2,3275	11,0010
		0,75	0	0,0427	0,1606	0,0277	3,0141	12,5190
	500	0,25	0	0,0912	0,2119	0,0341	5,4190	9,6789
		0,50	0	0,0422	0,1318	0,0219	9,1320	11,3170
		0,75	0	0,0112	0,0724	0,0115	11,8633	12,9115
30	100	0,25	0	0,3525	1,6721	0,3045	0,4357	8,9992
		0,50	0	0,2461	0,8222	0,1578	0,7693	10,4958
		0,75	3	0,0000	0,4196	0,0915	0,9946	12,3791
	250	0,25	0	0,3462	1,2995	0,1511	2,9955	9,1575
		0,50	0	0,3787	0,7085	0,0791	5,1511	10,7700
		0,75	0	0,2014	0,3621	0,0210	6,6270	12,6528
	500	0,25	0	0,2964	0,9596	0,0820	11,9596	9,3770
		0,50	0	0,2579	0,4907	0,0318	20,4519	11,0352
		0,75	0	0,1744	0,2739	0,0231	26,1625	12,9561

Tabela 7.6: Resultados da implementação usando GPGPU do híbrido de RNA e GRASP para as instâncias de GK.

		RNA+GRASP GPGPU					
m	n	Ótimos	*Gap* Mínimo	*Gap* Médio	DP	Tempo(s)	*Speedup*
15	100	0	0,2655	0,5019	0,1038	0,5321	10,0866
25	100	0	0,5558	0,7571	0,0876	0,7957	9,8542
25	150	0	0,6011	0,8828	0,1066	1,7684	9,9270
50	150	0	0,5375	0,8046	0,1066	3,1978	9,8218
25	200	0	0,3705	0,5006	0,0607	2,9934	10,4067
50	200	0	0,6648	0,9420	0,1037	5,6958	9,7624
25	500	0	0,2030	0,2777	0,0328	18,5499	10,5651
50	500	0	0,4467	0,5039	0,0314	33,6023	10,4154
25	1500	0	0,1016	0,1375	0,0154	164,6046	10,7695
50	1500	0	0,3508	0,3927	0,0221	307,5002	10,3350
100	2500	0	0,2762	0,3034	0,0127	1626,8087	10,2492

O desempenho promissor da versão usando GPGPU da heurística híbrida proposta pode ser notada no gráfico ilustrado na Figura 7.1, que ilustra a comparação dos *gaps* alcançados para as instâncias de ORLIB, agrupadas por número de itens, das implementações híbridas propostas com os obtidos pela rede neural aumentada apresentada no Capítulo 4 e pela de Deane e Agarwal [22] (RNA-DA), pela abordagem neurogenética também de Deane e Agarwal [23], pelo algoritmo de otimização por colônia (OCF) de formigas proposto Fingler *et al.* [28] e pelo algoritmo genético (AG-CB) de Chu e Beasley [8].

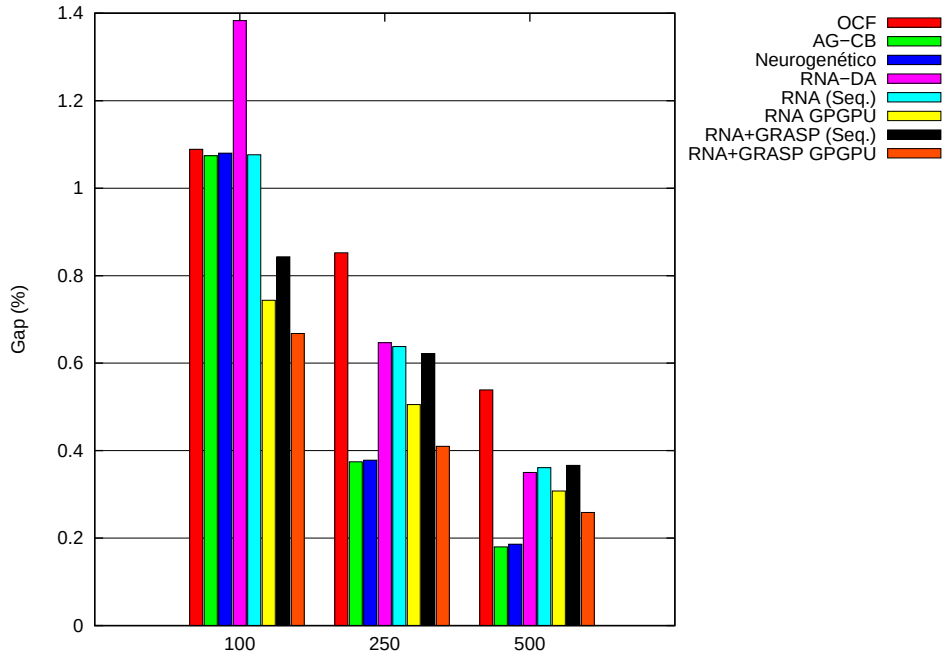


Figura 7.1: Comparação dos *gaps* obtidos para as instâncias de ORLIB.

Como pode ser visto no gráfico, as implementações híbridas atingiram resultados de boa qualidade em todas as configurações, especialmente, nas contendo 100 itens; com

250 ou 500 itens, os *gaps* foram ligeiramente maiores do que os obtidos pelo algoritmo genético e pela abordagem neurogenética.

A Figura 7.2 ilustra o gráfico comparativo dos tempos de execução da abordagem híbrida comparada aos das demais implementações, exceto aos do algoritmo genético, cujo tempo de execução apresentado no trabalho de referência é muito maior do que os demais, e aos do algoritmo neurogenético, cujas informações não estavam disponíveis no trabalho de referência.

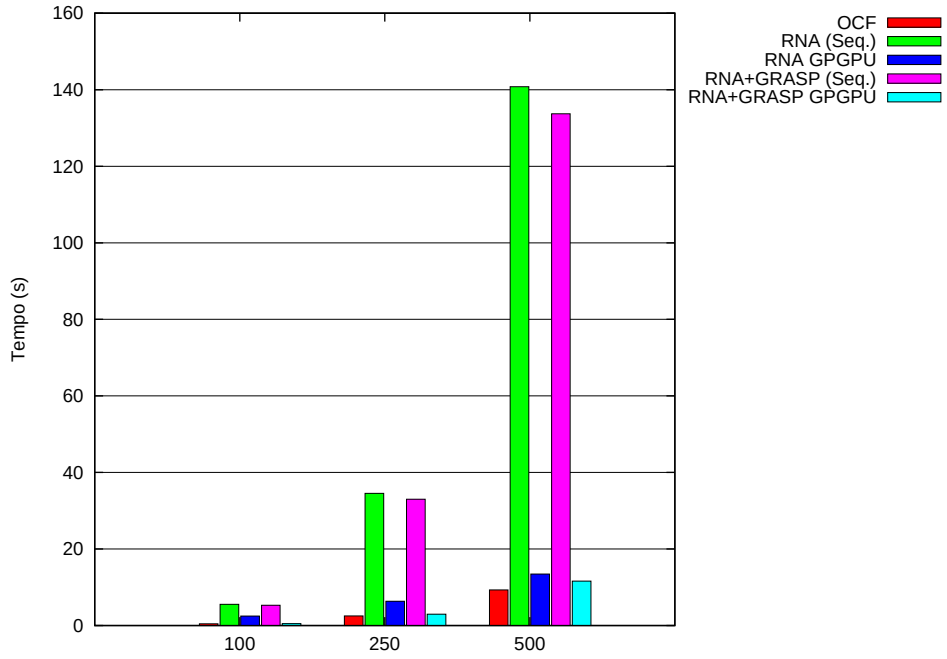


Figura 7.2: Comparação dos tempos obtidos para as instâncias de ORLIB.

A aplicação do teste de Wilcoxon de soma de *ranks* mostrou que os resultados obtidos para todos os conjuntos de testes não são significativamente diferentes dos valores de referência. Para o conjunto SAC-94, os p-valores calculados para os resultados sequenciais e paralelos foram 0,4353 e 0,4077, respectivamente; para as instâncias de GK, os valores de W foram iguais a 121 para a versão sequencial e para a paralela. A Tabela 7.7 mostra os p-valores calculados para as configurações das instâncias de ORLIB.

Tabela 7.7: p-valores calculados para os resultados do híbrido de RNA e GRASP para cada configuração de testes de ORLIB.

		p-valor	
m	n	Sequencial	GPGPU
5	100	0,3669	0,3922
	250	0,3558	0,3837
	500	0,3837	0,3894
10	100	0,2722	0,3050
	250	0,2722	0,3234
	500	0,3341	0,3287
30	100	0,2039	0,2254
	250	0,1127	0,1275
	500	0,1469	0,1370

7.4 Contribuições

As implementações da abordagem híbrida de RNA e GRASP descritas neste capítulo levaram contribuições relevantes, podendo-se citar:

- Desenvolvimento de uma heurística híbrida de RNA e GRASP;
- Publicação de artigo em *16th International Conference on Computational Science and Its Applications* (ICCSA 2016) [21];
- Constatação de que a associação de RCL melhorou resultados de RNA nas instâncias maiores;
- Proposição de nova alternativa para paralelização de RNA.

CAPÍTULO 8

Análise Comparativa

Como visto ao longo dos capítulos anteriores, as metaheurísticas podem ser utilizadas como alternativas válidas para resolver o problema da mochila multidimensional, ainda que de forma aproximada. Neste capítulo, é feita uma análise do comportamento observado nos diferentes conjuntos de teste, através da comparação das qualidades das soluções obtidas e dos tempos de execução de cada uma das implementações propostas. Além disso, visando mostrar que os resultados obtidos são competitivos, apresenta-se um comparativo com outras estratégias encontradas na literatura para a solução do problema.

Os gráficos das Figuras 8.1 e 8.2 ilustram, respectivamente, os *gaps* e os tempos de execução obtidos pelas metaheurísticas implementadas para as instâncias de SAC-94.

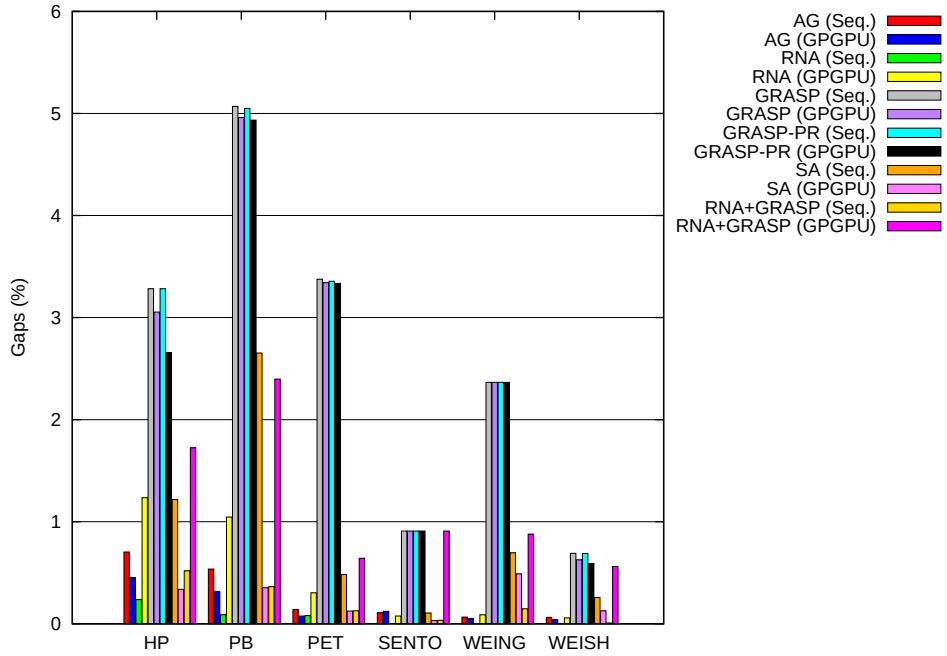


Figura 8.1: *Gaps* obtidos para as instâncias de SAC-94.

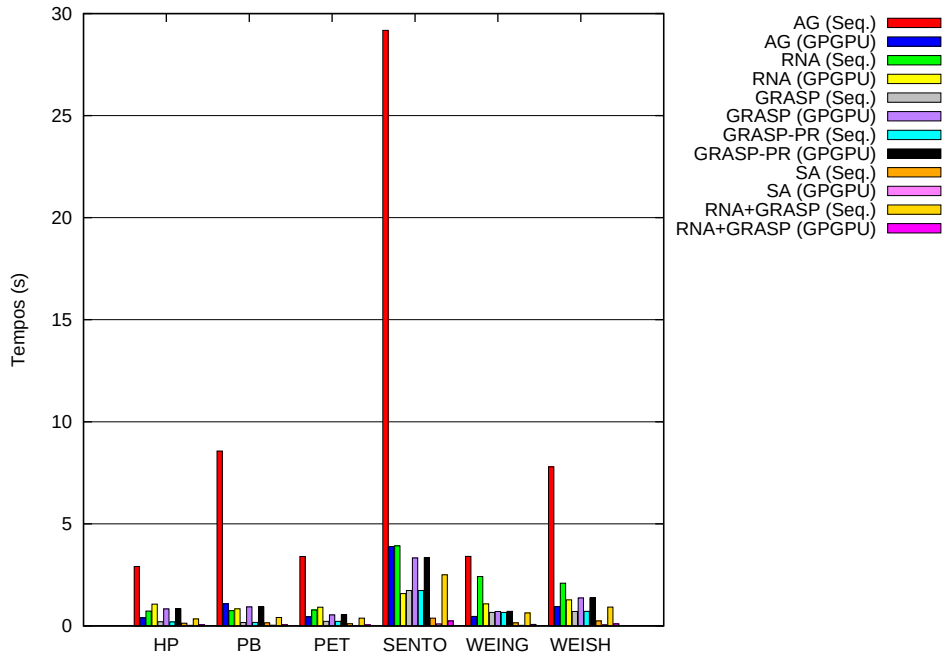


Figura 8.2: Tempos de execução obtidos para as instâncias de SAC-94.

Como pode-se perceber pelo gráfico, a rede neural aumentada em sua implementação sequencial foi a abordagem que levou aos melhores resultados, com *gaps* próximos a zero em todos os subconjuntos, enquanto as diversas variantes de GRASP não conseguiram alcançar soluções tão boas quanto as demais metaheurísticas. Isso mostra que, quando as instâncias a serem resolvidas são pequenas, as estratégias de aprendizado são capazes de fazer com que a busca pela solução seja melhor direcionada, em especial, quando associada à estratégia gulosa de sempre escolher o melhor elemento a ser adicionado, como ocorre com a rede neural aumentada. A associação da construção da RCL ao processo de escolha de tal item não foi vantajosa nas instâncias reduzidas de SAC-94; essa situação é condizente com o baixo desempenho apresentado por GRASP.

Outra situação que merece destaque está relacionada ao fato de que os algoritmos genéticos, que não levaram a bons resultados de uma maneira geral (como visto no Capítulo 3), neste caso específico, foram eficientes na obtenção de soluções de boa qualidade, com *gaps* muito próximos a zero em quatro dos seis subconjuntos. Essa característica leva, novamente, a crer que estratégias gulosas são mais adequadas às instâncias pequenas. Isso porque, na implementação de algoritmos genéticos descritos, os dois melhores indivíduos da população são sempre escolhidos para dar origem aos indivíduos da nova geração, fazendo com que as características das soluções de melhor qualidade se mantenham.

É importante salientar que, apesar da vantagem da rede neural aumentada sequencial, todas as abordagens apresentaram resultados com qualidade menos de 2% inferiores às soluções ótimas, exceto as versões de GRASP e a versão paralela do híbrido RNA+GRASP. Com relação aos tempos de execução, apenas a implementação sequencial de algoritmos genéticos apresentou valores muito superiores aos demais, tornando sua aplicação menos interessante.

À medida que as dimensões do problema aumentam, o comportamento dos algoritmos se altera, como pode ser notado pelos gráficos apresentados nas Figuras 8.3, 8.5,

8.7, 8.9, 8.11, 8.13, 8.15, 8.17 e 8.19 que comparam os *gaps* obtidos paras as diferentes configurações das instâncias de ORLIB; os tempos de execução são mostrados nas Figuras 8.4, 8.6, 8.8, 8.10, 8.12, 8.14, 8.16, 8.18 e 8.20.

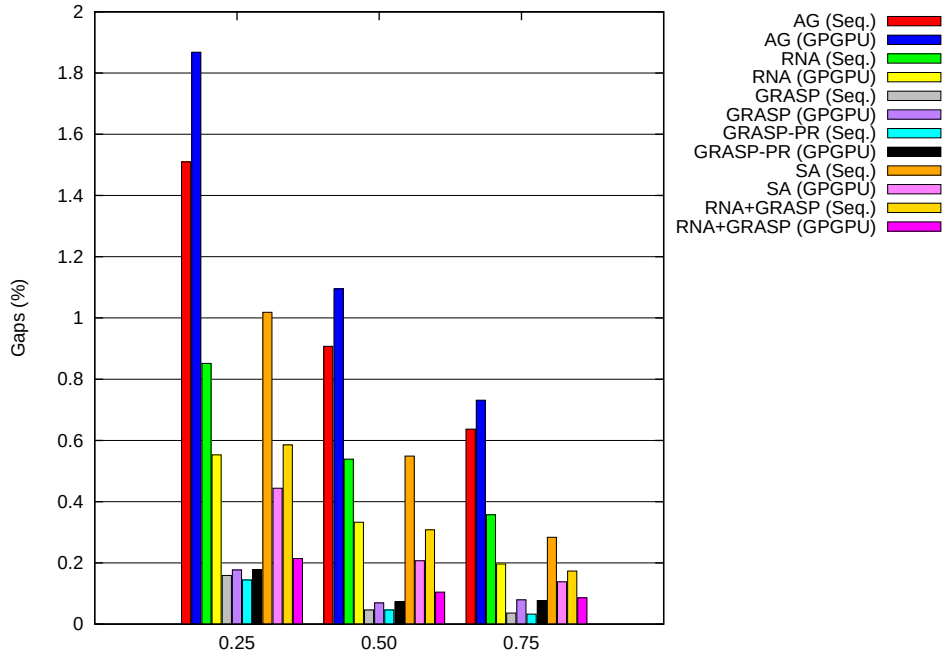


Figura 8.3: *Gaps* obtidos para as instâncias com 5 recursos e 100 itens de ORLIB.

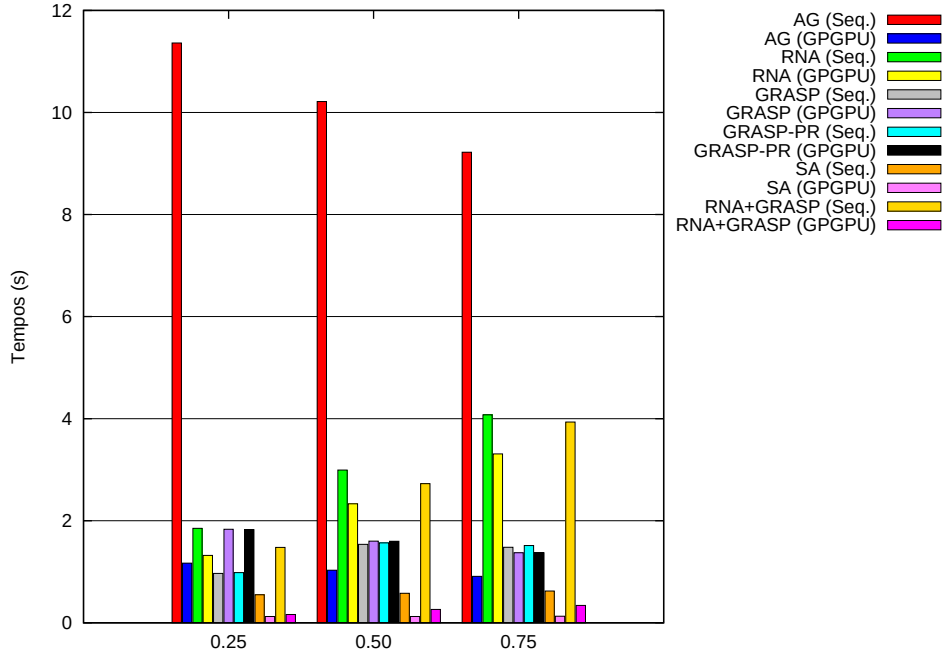


Figura 8.4: Tempos de execução obtidos para as instâncias com 5 recursos e 100 itens de ORLIB.

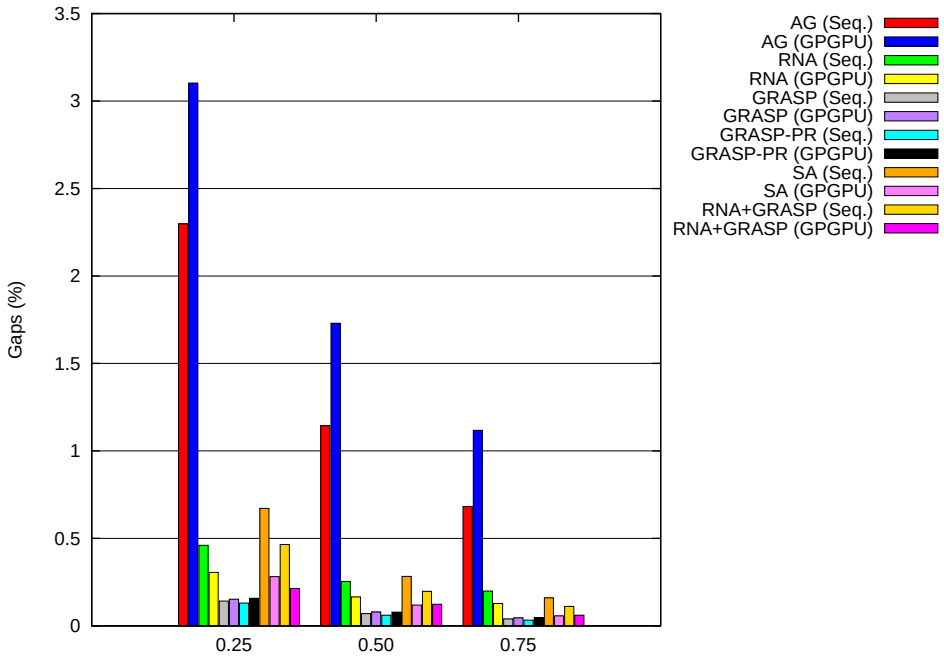


Figura 8.5: *Gaps* obtidos para as instâncias com 5 recursos e 250 itens de ORLIB.

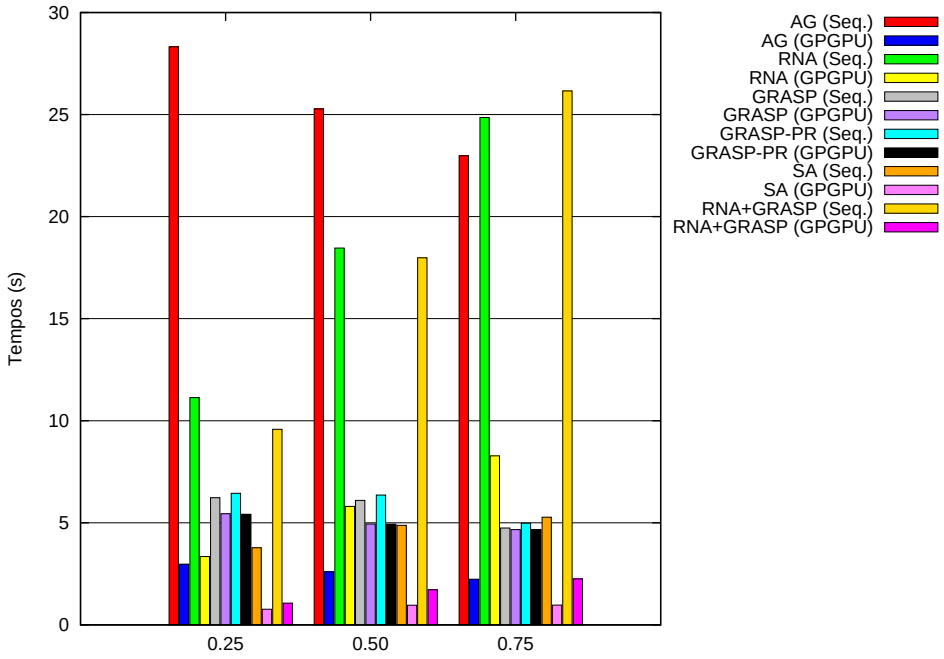


Figura 8.6: Tempos de execução obtidos para as instâncias com 5 recursos e 250 itens de ORLIB.

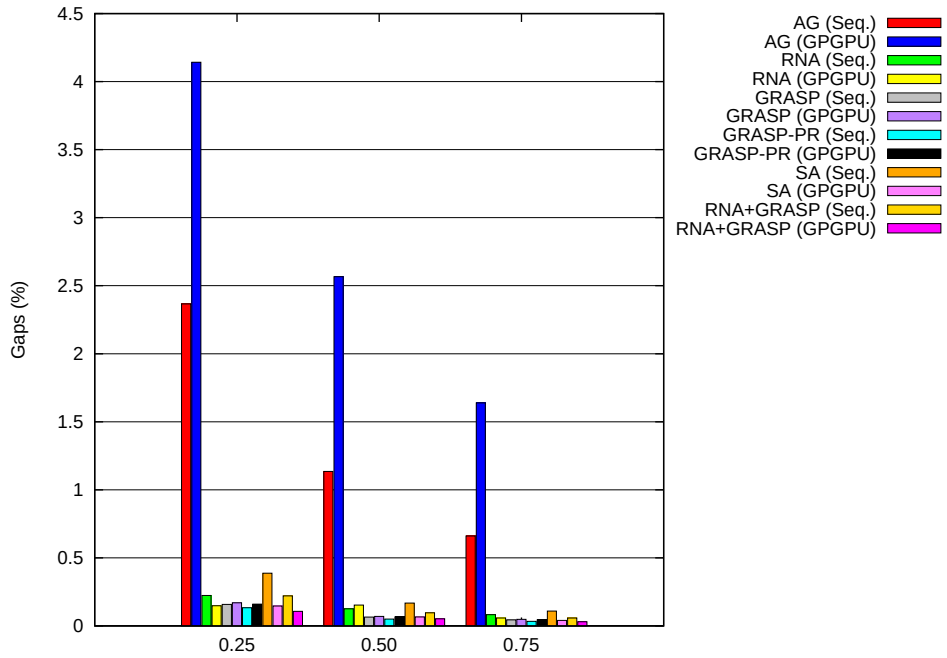


Figura 8.7: *Gaps* obtidos para as instâncias com 5 recursos e 500 itens de ORLIB.

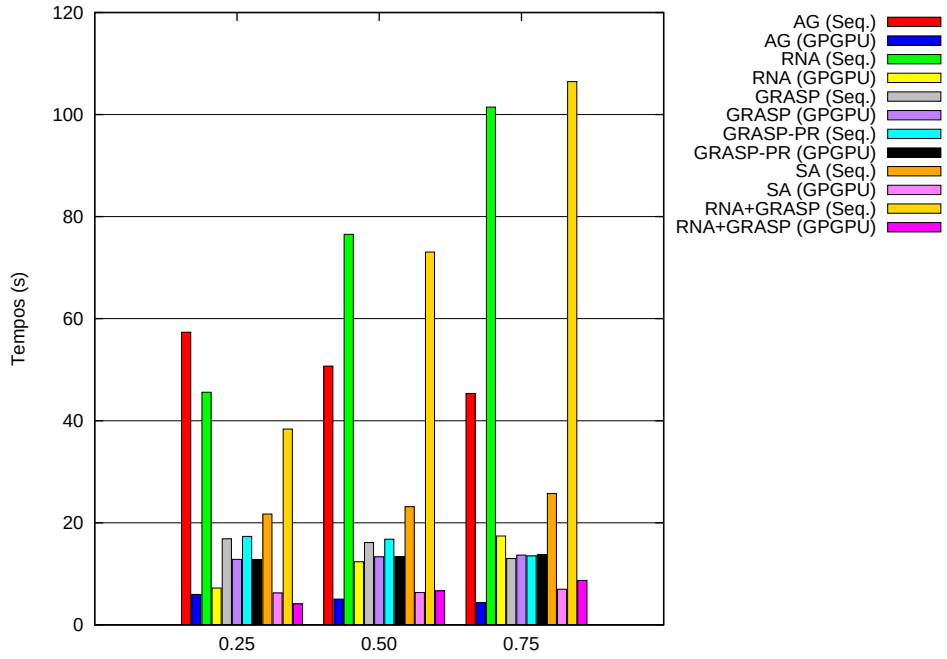


Figura 8.8: Tempos de execução obtidos para as instâncias com 5 recursos e 500 itens de ORLIB.

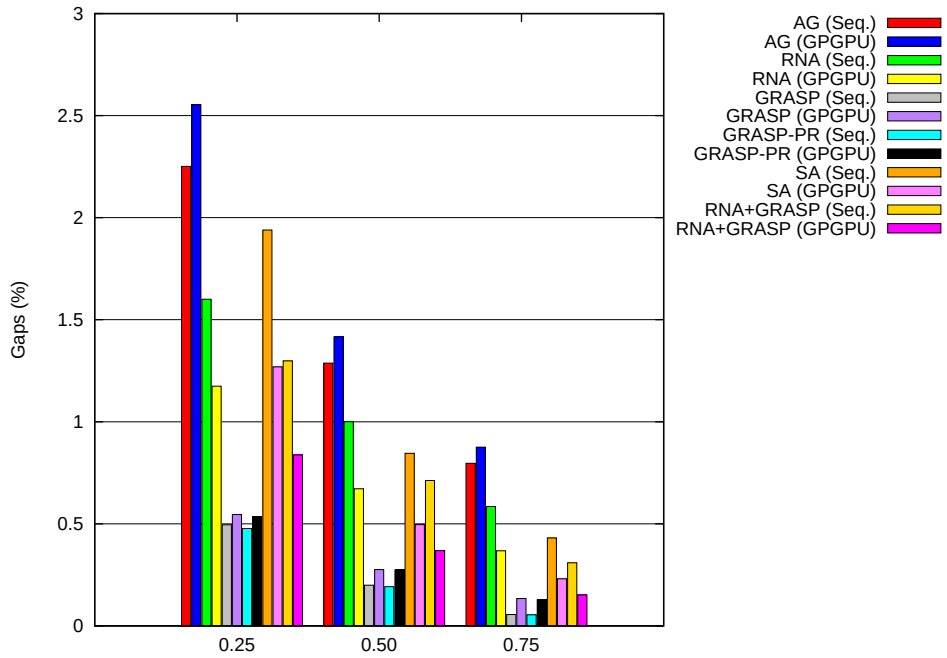


Figura 8.9: *Gaps* obtidos para as instâncias com 10 recursos e 100 itens de ORLIB.

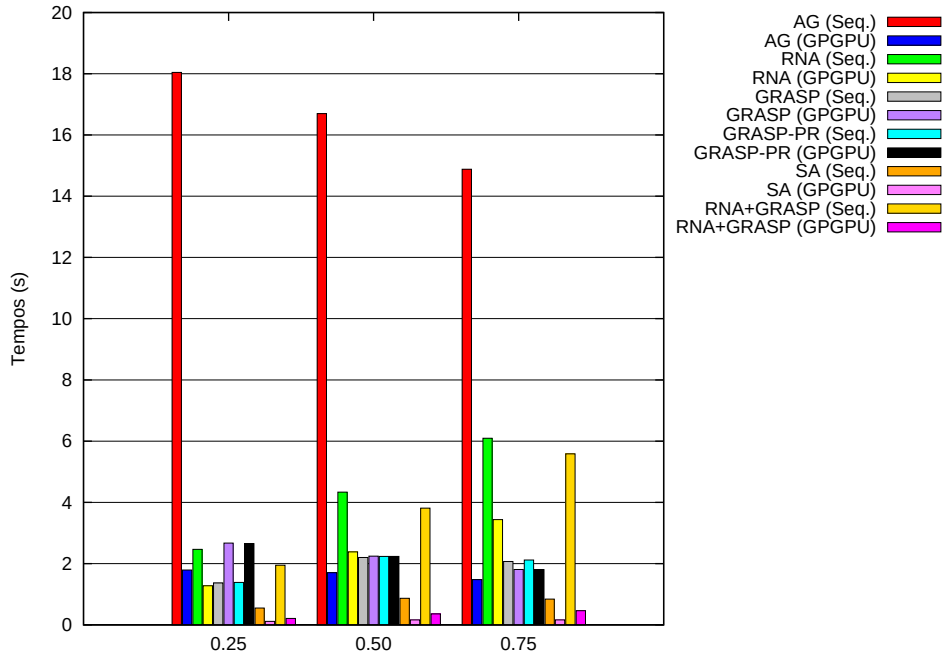


Figura 8.10: Tempos de execução obtidos para as instâncias com 10 recursos e 100 itens de ORLIB.

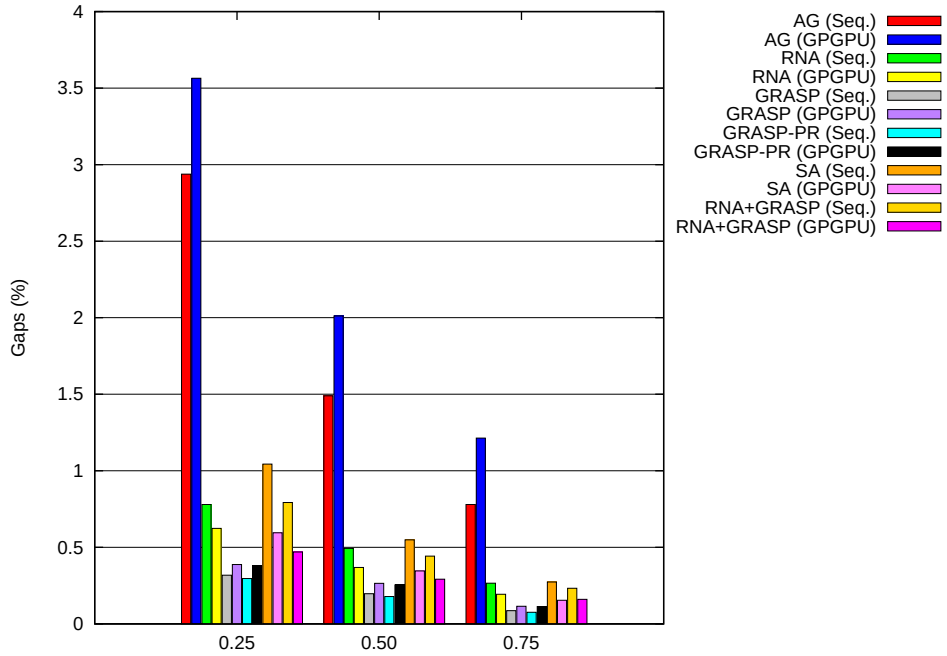


Figura 8.11: *Gaps* obtidos para as instâncias com 10 recursos e 250 itens de ORLIB.

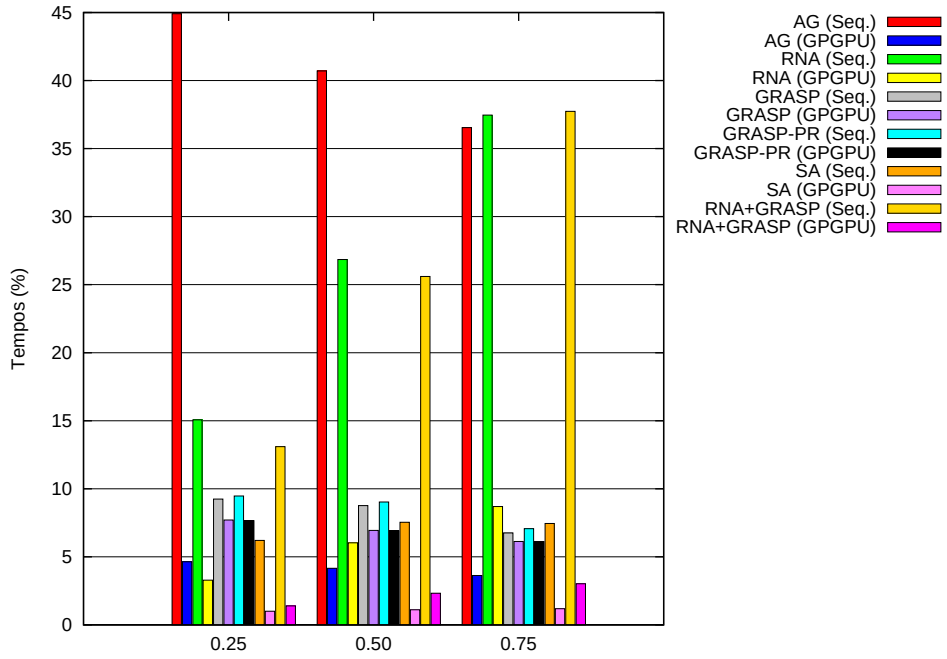


Figura 8.12: Tempos de execução obtidos para as instâncias com 10 recursos e 250 itens de ORLIB.

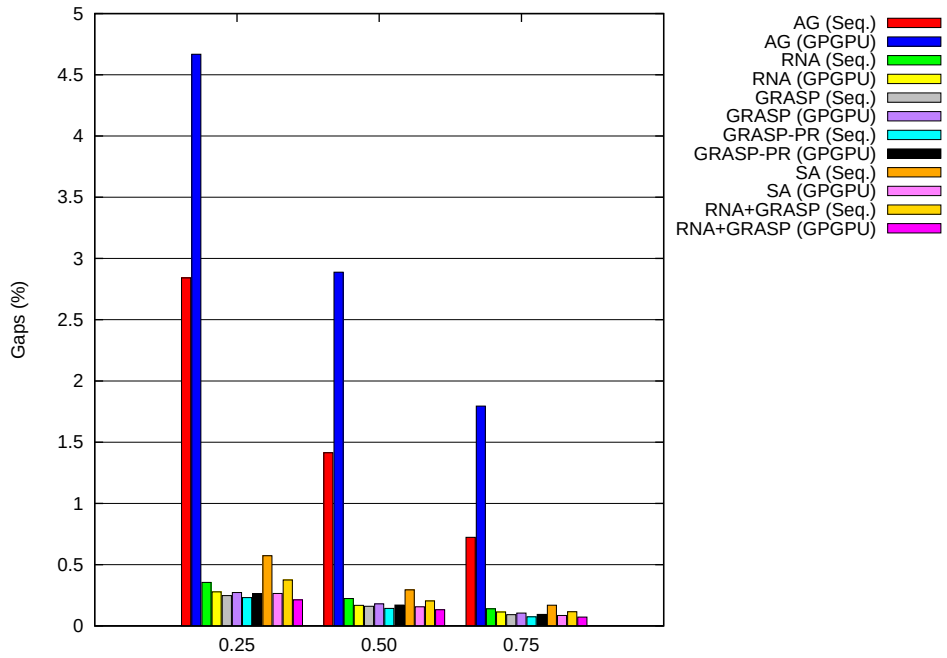


Figura 8.13: *Gaps* obtidos para as instâncias com 10 recursos e 500 itens de ORLIB.

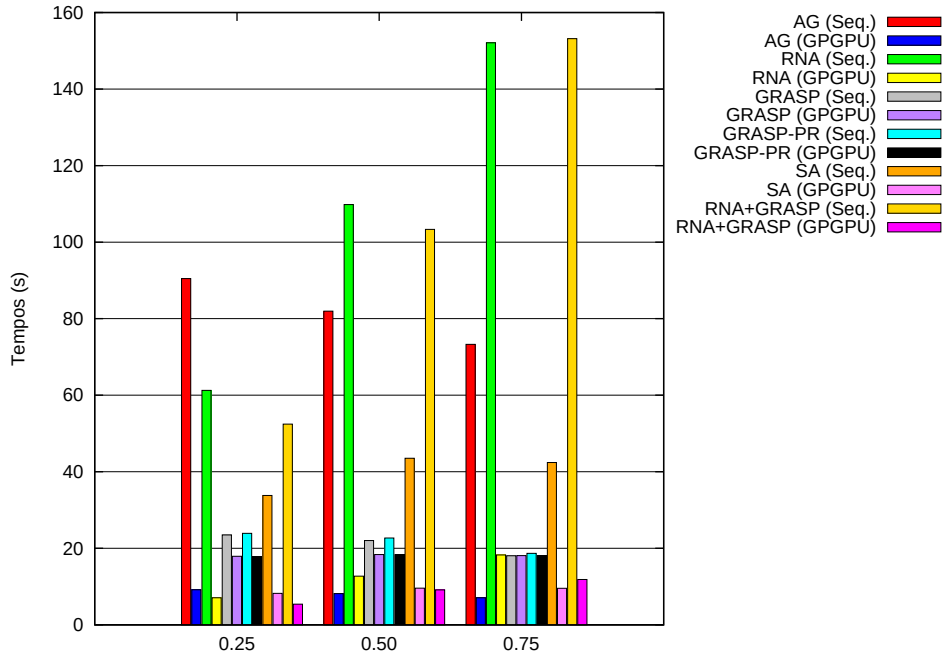


Figura 8.14: Tempos de execução obtidos para as instâncias com 10 recursos e 500 itens de ORLIB.

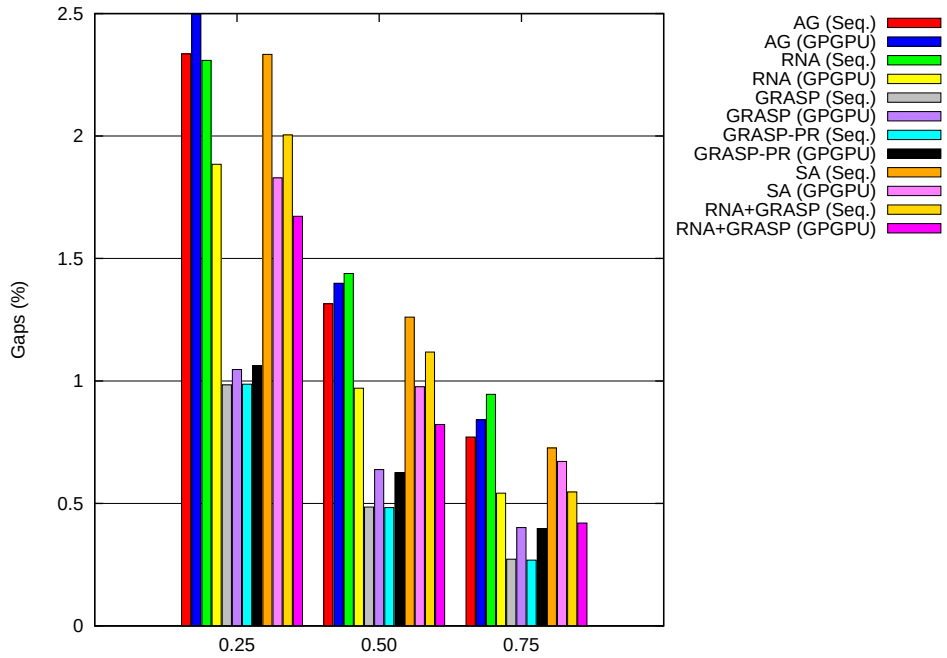


Figura 8.15: *Gaps* obtidos para as instâncias com 30 recursos e 100 itens de ORLIB.

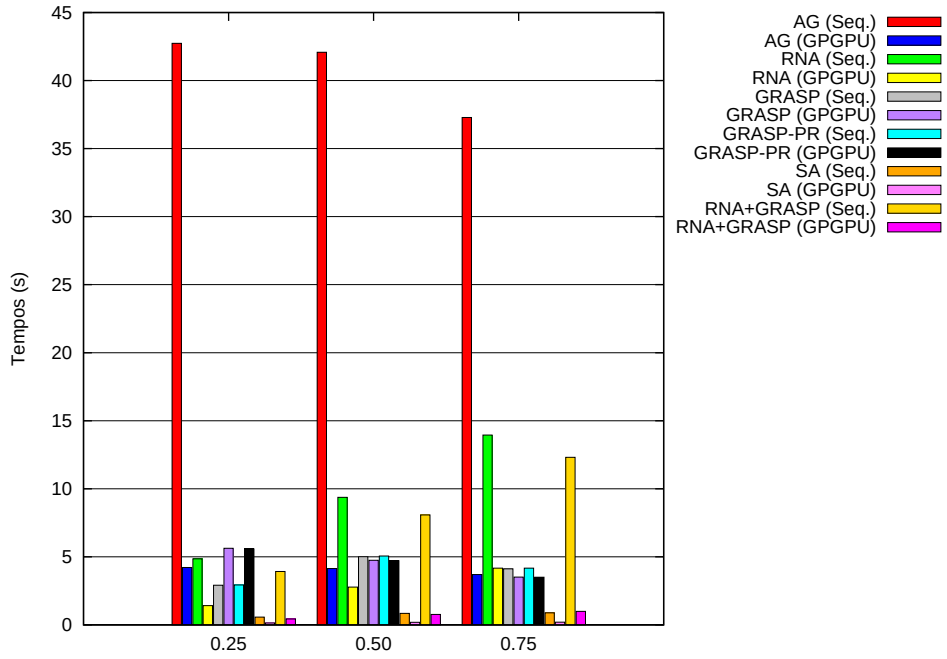


Figura 8.16: Tempos de execução obtidos para as instâncias com 30 recursos e 100 itens de ORLIB.

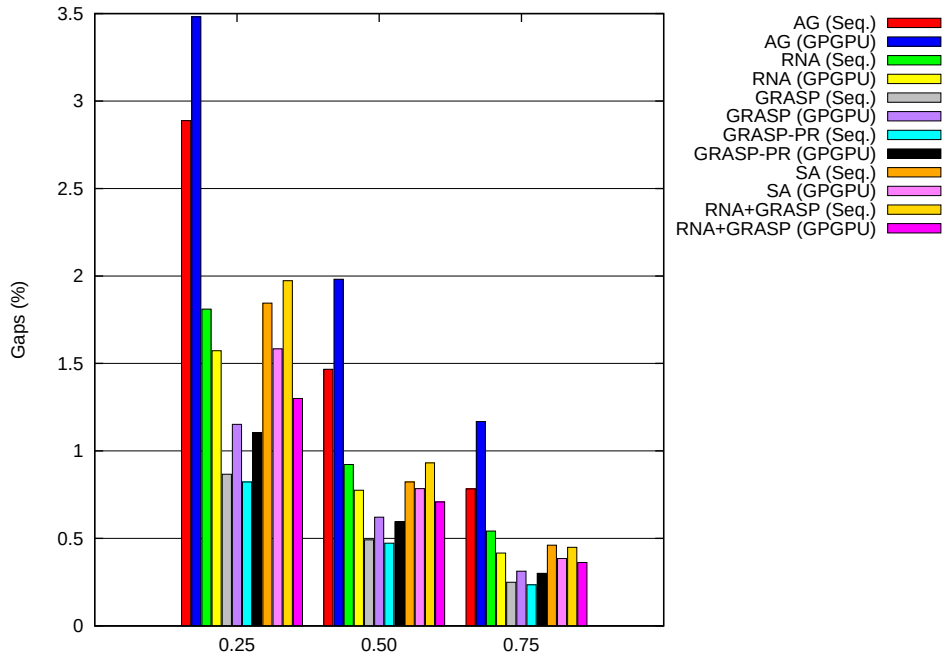


Figura 8.17: *Gaps* obtidos para as instâncias com 30 recursos e 250 itens de ORLIB.

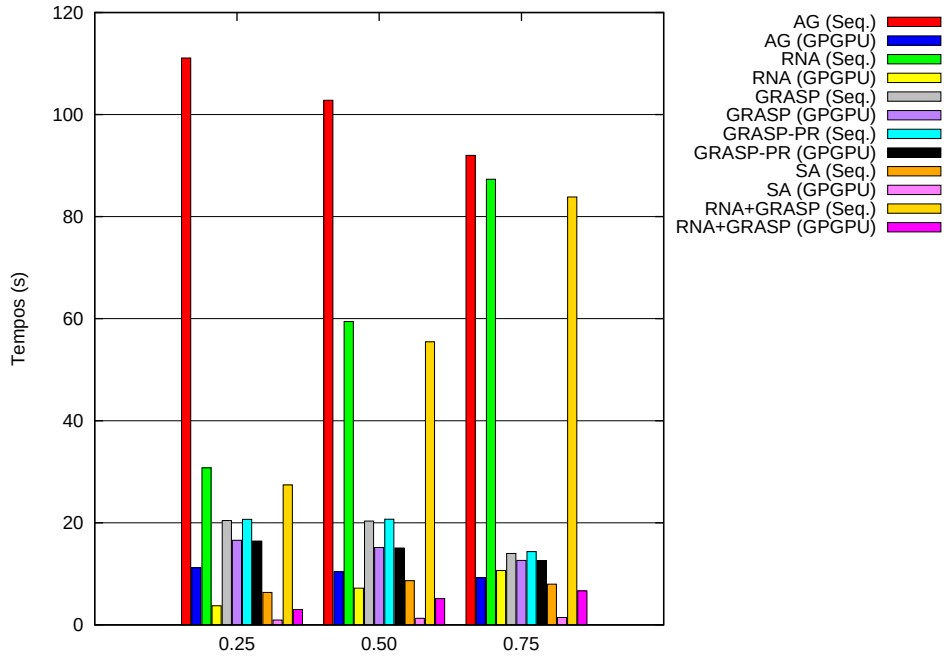


Figura 8.18: Tempos de execução obtidos para as instâncias com 30 recursos e 250 itens de ORLIB.

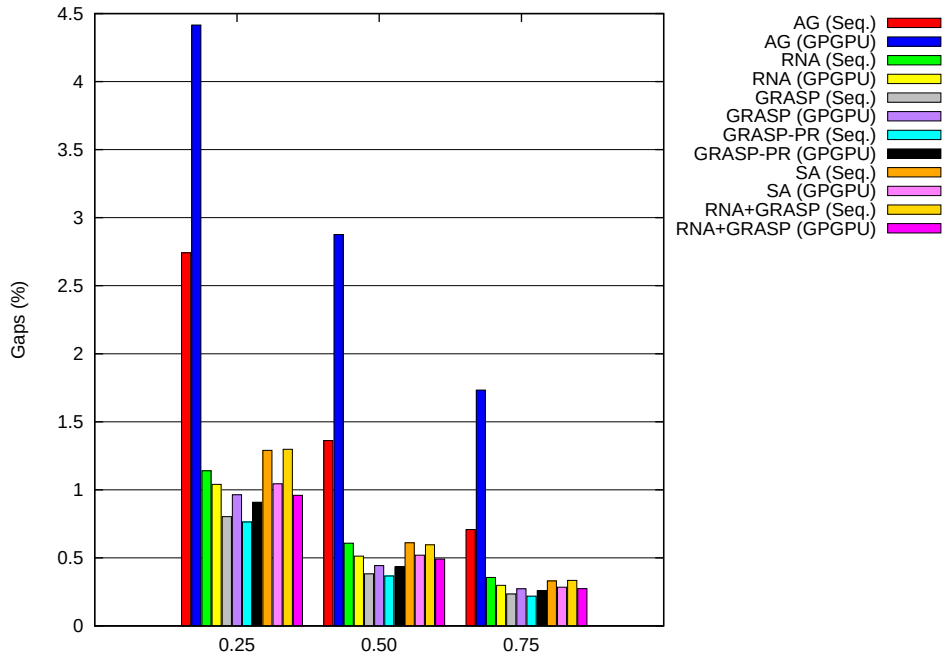


Figura 8.19: *Gaps* obtidos para as instâncias com 30 recursos e 500 itens de ORLIB.

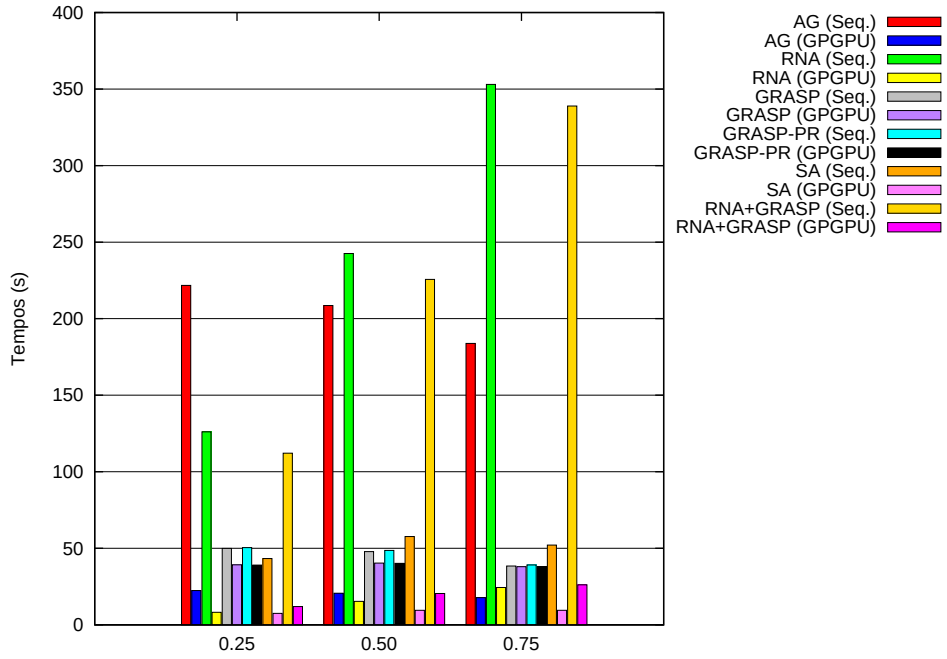


Figura 8.20: Tempos de execução obtidos para as instâncias com 30 recursos e 500 itens de ORLIB.

A qualidade dos resultados alcançados pelas metaheurísticas com as instâncias de ORLIB foi significativamente diferente da percebida com as de SAC-94. Com essas instâncias, as variantes de GRASP levaram aos menores *gaps*, em especial a implementação sequencial com o uso da intensificação por *path-relinking*. Essa situação mostra que com instâncias maiores, o uso de alternativas que possibilitem variabilidade, ainda que limitada, na escolha dos itens a serem adicionados possibilita que a

busca por soluções seja mais ampla, escapando dos máximos locais com maior facilidade. Ratificando essa teoria, nota-se que a abordagem híbrida, que faz uso da RCL para construção de suas soluções, obteve resultados de melhor qualidade do que os alcançados pela rede neural aumentada pura.

As soluções de menor qualidade foram obtidas pelos algoritmos genéticos, o que reforça a ideia de que estratégias totalmente gulosas não conseguem explorar o espaço de soluções de maneira eficiente. À medida que o tamanho do problema aumenta, os algoritmos gulosos ficam presos à uma determinada região do espaço de busca, explorando-a exaustivamente, deixando, entretanto de visitar outras regiões.

Um aspecto percebido claramente é que, com as instâncias de SAC-94, o uso do paralelismo levou a melhores resultados com algoritmos genéticos, GRASP e *simulated annealing*, enquanto as outras heurísticas obtiveram soluções de melhor qualidade com sua versão sequencial. Esse panorama se inverteu com as instâncias de ORLIB, com a exceção de *simulated annealing*, cujos menores *gaps* foram obtidos pela versão paralela com ambos conjuntos de teste.

Quanto aos tempos de execução, nota-se que as implementações sequenciais de algoritmos genéticos e redes neurais aumentadas, tanto a versão pura quanto a hibridizada com GRASP, foram as mais demoradas em todas as configurações de ORLIB, com pequenas variações entre elas. *Simulated annealing* em sua versão paralela, por sua vez, foi a heurística que teve os menores tempos de execução, seguida pela implementação paralela da abordagem híbrida. Vale destacar que, à medida que o número de itens aumenta, os *gaps* alcançados por *simulated annealing* se tornam muito próximos aos obtidos pelas variantes de GRASP, o que torna seu o uso bastante interessante.

Com o conjunto GK, cujas instâncias contêm entre 15 e 100 recursos e entre 100 e 2500 itens, os melhores resultados foram alcançados por diferentes algoritmos (versão sequencial da proposta híbrida e versões usando GPGPU da rede neural aumentada e de *simulated annealing*), enquanto ambas implementações de algoritmos genéticos levaram aos maiores *gaps*. As demais metaheurísticas levaram a soluções de qualidades similares, todas com qualidade menos de 2% inferiores às soluções de referência. Essa situação pode ser constatada na Tabela 8.1, que mostra os *gaps* das soluções obtidas por cada metaheurística com relação às alcançadas pelo CPLEX; a Tabela 8.2 mostra os tempos de execução necessários. Em ambas tabelas, os melhores valores estão destacados em azul e os piores em vermelho.

Tabela 8.1: *Gaps* obtidos pelas metaheurísticas para as instâncias de GK.

m	n	AG(S)	AG(G)	RNA(S)	RNA(G)	GRASP(S)	GRASP(G)	GRASP+PR(S)	GRASP+PR(G)	SA(S)	SA(G)	RNA+GRASP(S)	RNA+GRASP(G)
15	100	1,2843	1,1790	0,6444	0,6205	0,2452	0,2567	0,2399	0,2452	0,4983	0,3514	0,7603	0,5019
25	100	1,2986	1,2119	1,1403	1,0241	0,8607	1,1445	0,6165	0,8127	0,9197	0,6864	1,2978	0,7571
25	150	1,4869	1,6272	1,0431	0,9889	1,1875	1,5405	0,9954	1,2482	0,9512	0,7402	1,4244	0,8828
50	150	1,4866	1,6727	1,0196	1,0057	0,6410	0,9254	0,5462	0,7936	1,0653	0,9352	1,3300	0,8046
25	200	1,4785	1,7114	0,7666	0,7031	0,2064	0,4918	0,1764	0,4014	0,6180	0,4358	0,8509	0,5006
50	200	1,4864	1,5311	0,9771	0,9428	1,4038	1,7827	1,1079	1,2826	1,0658	0,7899	1,4347	0,9420
25	500	1,7139	2,1872	0,2335	0,1990	0,3608	0,4699	0,2996	0,3577	0,4602	0,2581	0,4358	0,2777
50	500	1,7312	2,0320	0,5229	0,5117	0,3217	0,4873	0,3036	0,4195	0,5909	0,4559	0,7260	0,5039
25	1500	2,0275	2,7742	0,0591	0,0541	1,1746	1,4297	0,9320	1,0858	0,2293	0,1184	0,2019	0,1375
50	1500	1,9122	2,2250	0,2135	0,2125	1,4431	1,7745	1,2086	1,3535	0,4543	0,3046	0,5105	0,39272
100	2500	1,6303	1,8433	0,3014	0,2884	0,2476	0,3610	0,2235	0,2689	0,5719	0,4263	0,4932	0,3034

Tabela 8.2: Tempos de execução das metaheurísticas para as instâncias de GK.

m	n	AG(S)	AG(G)	RNA(S)	RNA(G)	GRASP(S)	GRASP(G)	GRASP+PR(S)	GRASP+PR(G)	SA(S)	SA(G)	RNA+GRASP(S)	RNA+GRASP(G)
15	100	19,9953	1,6144	8,7956	2,4894	2,1534	2,2903	2,1613	2,2934	0,4109	0,1112	5,3671	0,5321
25	100	30,1836	2,3940	11,5195	2,7272	3,6371	3,2597	3,6411	3,2913	0,3516	0,1186	7,8410	0,7957
25	150	45,5740	3,7919	27,5434	4,1436	8,0688	5,6600	8,1372	5,7151	1,1744	0,2925	17,5549	1,7684
50	150	85,1817	7,2160	42,3335	5,1283	14,2126	9,3919	14,2145	9,4217	1,0209	0,3974	31,4080	3,1978
25	200	60,5840	4,9241	52,0517	5,5348	15,0124	8,5644	15,1128	8,5730	1,9129	0,5820	31,1515	2,9934
50	200	113,0850	9,6163	77,9059	6,8048	23,3570	14,5508	23,5639	14,7599	1,7390	0,7860	55,6047	5,6958
25	500	150,9935	13,2084	419,8423	14,8004	89,0985	33,0904	90,9779	33,3846	18,6645	5,7002	195,9820	18,5499
50	500	282,6370	27,7136	583,4163	18,2666	160,2657	54,2252	162,1004	54,3761	18,7049	6,9119	349,9812	33,6023
25	1500	459,5097	44,6006	6349,4797	55,0975	387,5510	219,5180	391,9932	223,2568	384,7354	117,4823	1772,7040	164,6046
50	1500	850,4931	94,3738	7789,9975	71,6803	674,3600	410,3496	680,9785	419,0884	360,3986	103,9691	3178,0086	307,5002
100	2500	2751,2437	391,5188	36731,5515	189,1680	2646,9373	1824,2332	2652,3866	1834,6083	1493,3889	380,6320	16673,4771	1626,8087

Vale notar que todas as melhores soluções foram menos que 1% inferiores às soluções obtidas pelo CPLEX e, ainda, que as três metaheurísticas com melhores soluções obtiveram resultados de qualidade próxima em praticamente todas as instâncias; com ligeira vantagem para *simulated annealing* que, na média, obteve o menor *gap*. *Simulated annealing* também apresentou os menores tempos de execução em 8 das 11 instâncias, entretanto, quando o número de itens é muito grande – a partir de 1500, nas últimas instâncias –, o tempo de execução cresce significativamente. Esse comportamento é esperado devido ao aumento da cadeia de Markov usada no algoritmo. Nas outras três instâncias, a rede neural aumentada paralela, obteve os menores tempos de execução, justificado pelo fato de seu comportamento ser menos afetado pelo aumento no número de itens da instância a ser resolvida.

Visando mostrar que os resultados das metaheurísticas implementadas são competitivos, eles foram comparados aos obtidos por outros trabalhos da bibliografia pesquisada. Para tal, os resultados das implementações que levaram às melhores soluções em cada uma das configurações são comparados aos das técnicas usadas em tais trabalhos. Os gráficos das Figuras 8.21 e 8.22 ilustram o comparativo dos *gaps* e dos tempos para as instâncias de SAC-94 e os gráficos das Figuras 8.23, 8.24, 8.25, 8.26, 8.27 e 8.28 mostram as comparações para as instâncias de ORLIB.

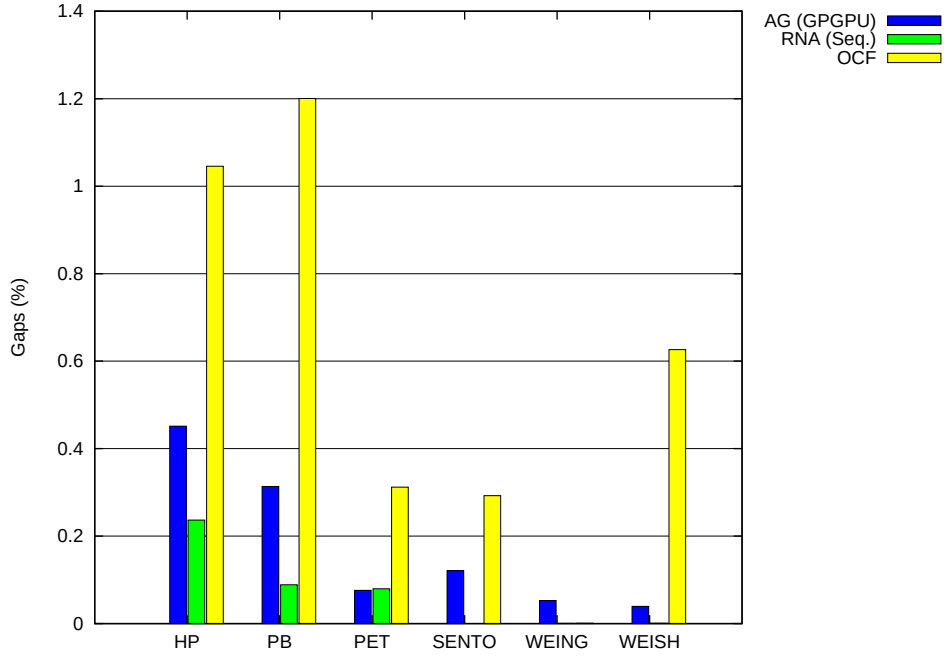


Figura 8.21: Comparação dos melhores *gaps* obtidos para as instâncias de SAC-94 comparados com trabalhos de referência.

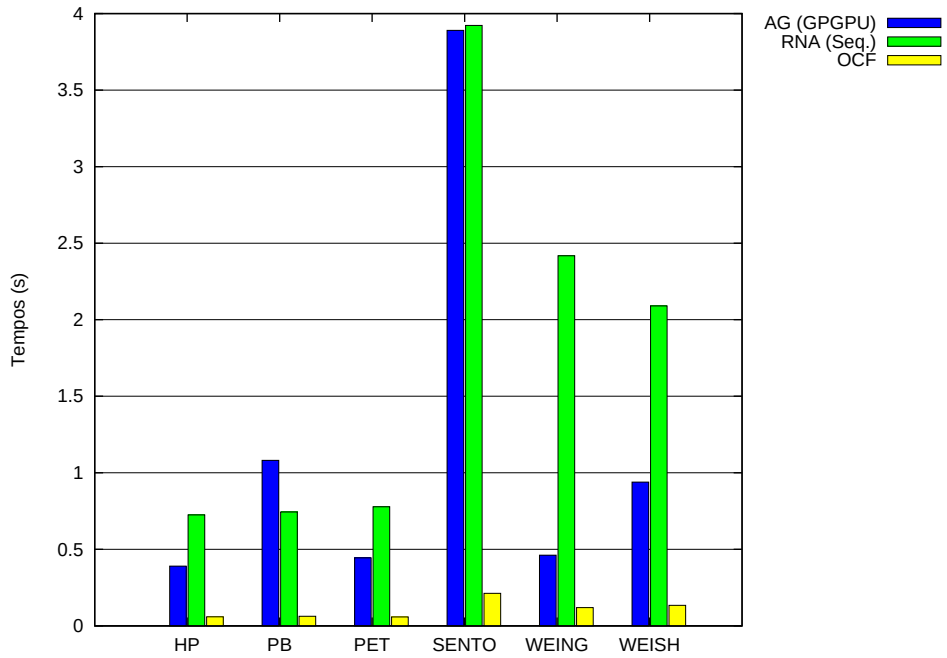


Figura 8.22: Comparação dos tempos obtidos para as instâncias de SAC-94 comparados com trabalhos de referência.

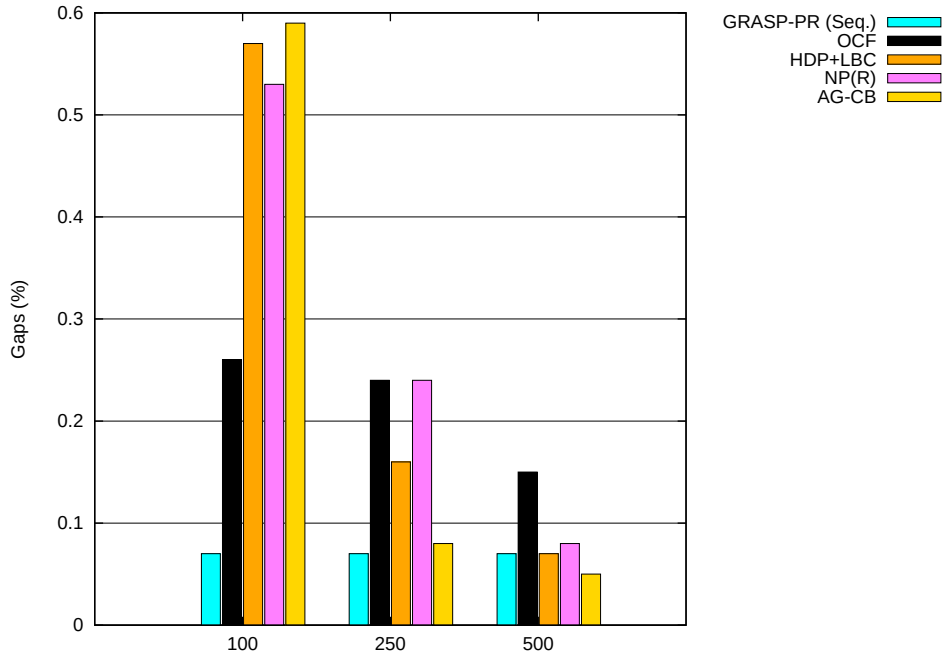


Figura 8.23: Comparação dos melhores *gaps* obtidos para as instâncias com 5 recursos de ORLIB comparados com trabalhos de referência.

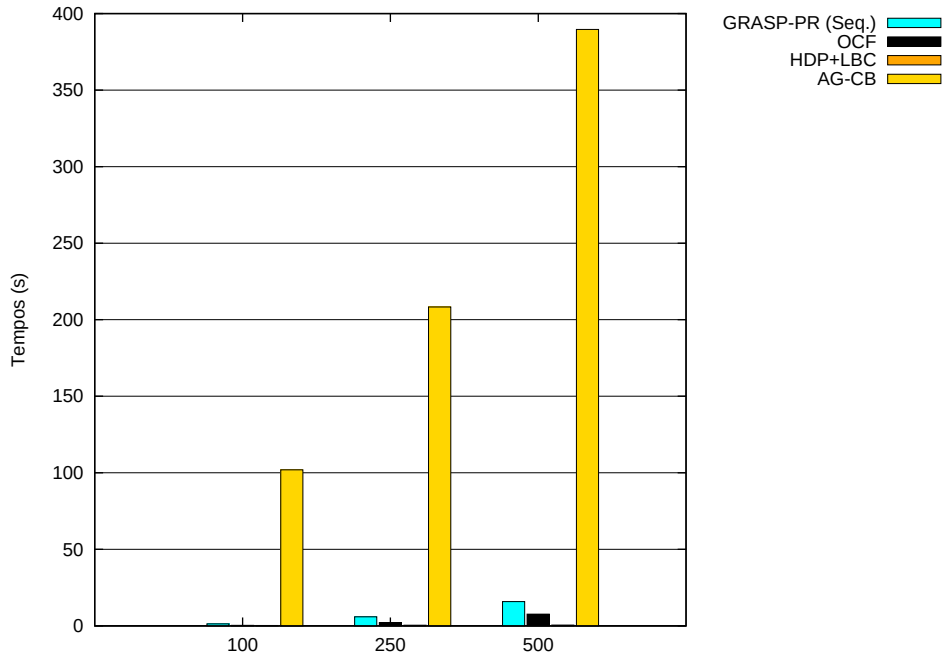


Figura 8.24: Comparação dos tempos obtidos para as instâncias com 5 recursos de ORLIB comparados com trabalhos de referência.

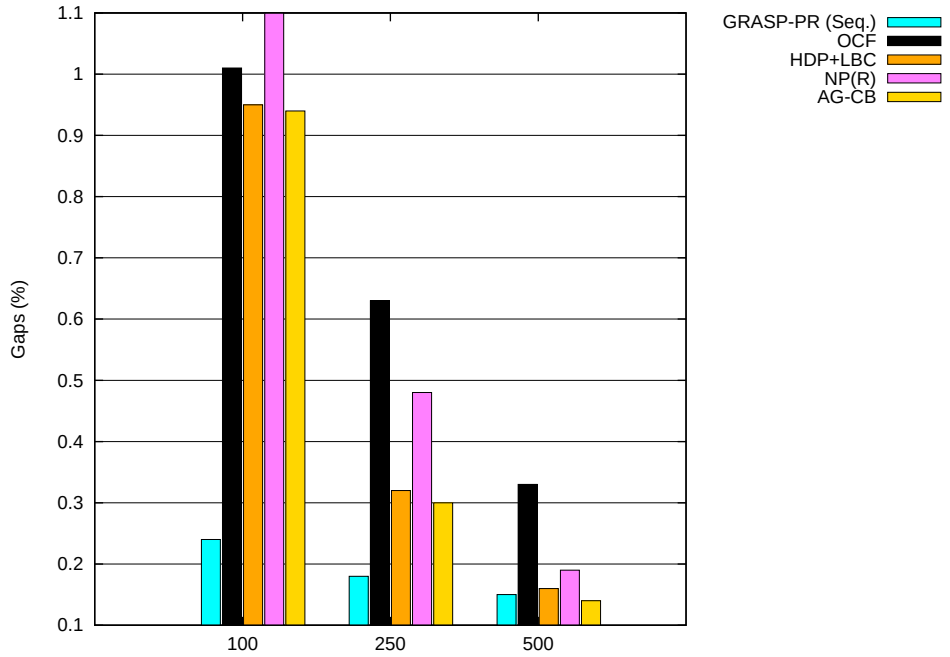


Figura 8.25: Comparação dos melhores *gaps* obtidos para as instâncias com 10 recursos de ORLIB comparados com trabalhos de referência.

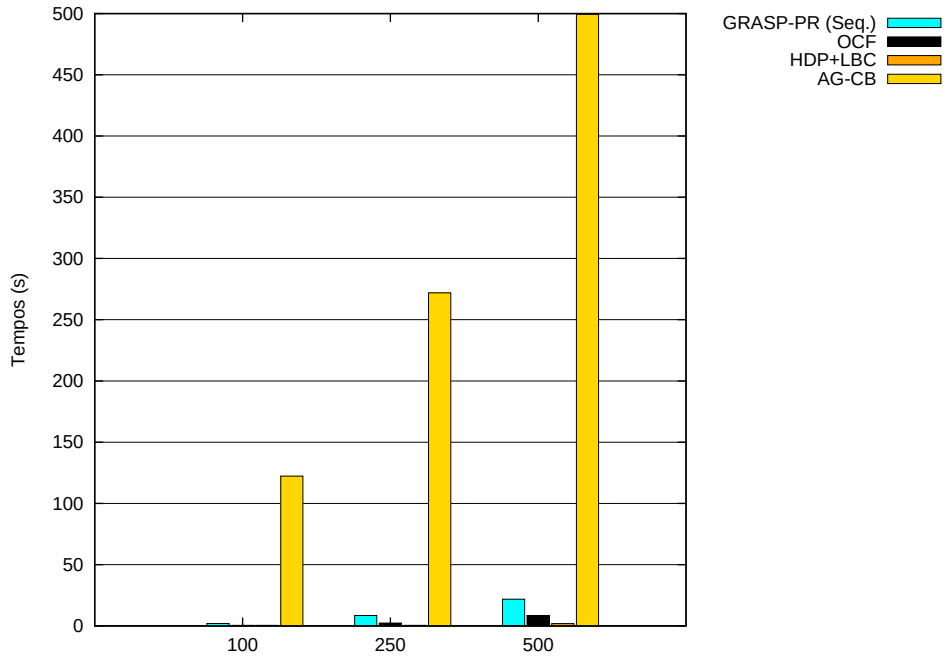


Figura 8.26: Comparação dos tempos obtidos para as instâncias com 10 recursos de ORLIB comparados com trabalhos de referência.

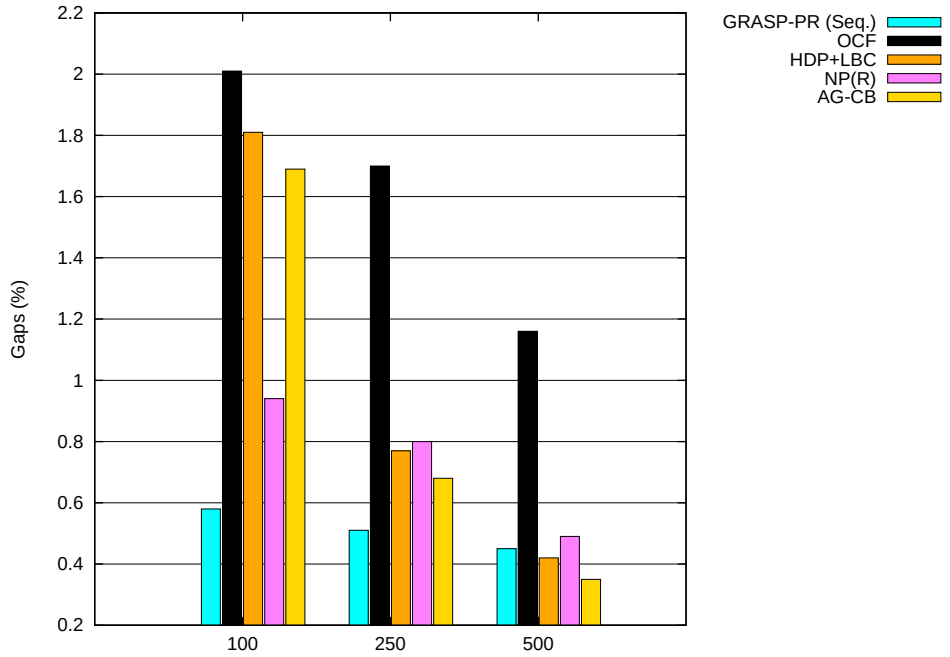


Figura 8.27: Comparação dos melhores *gaps* obtidos para as instâncias com 30 recursos de ORLIB comparados com trabalhos de referência.

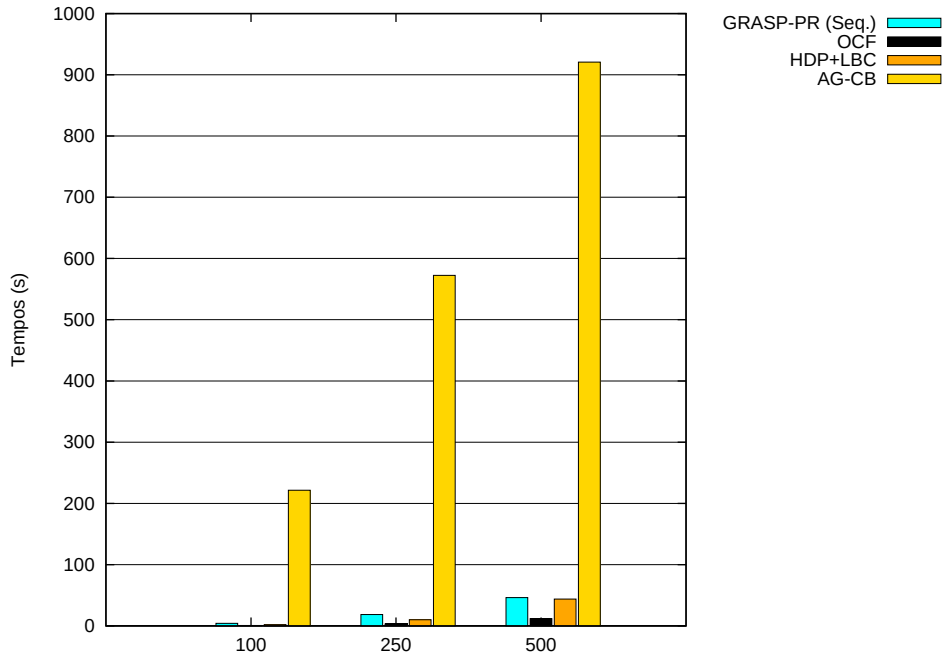


Figura 8.28: Comparação dos tempos obtidos para as instâncias com 30 recursos de ORLIB comparados com trabalhos de referência.

Para as instâncias de SAC-94 os resultados da implementação sequencial de redes neurais aumentadas (RNA (Seq.)) e da implementação paralela de algoritmos genéticos (AG (GPGPU)) foram comparados aos alcançados pela otimização por colônia de formigas (OCF) de Fingler *et al.* [29]. Os gráficos mostram que os *gaps* da rede neural aumentada e do algoritmo genético são bem menores que os obtidos pela colônia de formigas em todos os subconjuntos, entretanto, para alcançar tais resultados, ambas as abordagens demandaram tempo de execução bastante maior.

Os resultados alcançados pela abordagem sequencial de GRASP com o uso de *path-relinking* superaram os obtidos pelas demais abordagens propostas em todas as configurações de ORLIB, por esse motivo, ela foi comparada à otimização por colônia de formigas (OCF) [29], ao método de programação dinâmica com melhoria nas computações de limite inferior (HDP + LBC) de Boyer, Elkihel e El Baz [11], ao novo método de redução de problemas de Hill, Cho e Moore (NP(R)) [35] e ao algoritmo genético de Chu e Beasley (AG-CB) [17]. Pela análise dos gráficos, pode-se notar que GRASP com *path-relinking* levou a soluções de qualidade superior em quase todas as comparações, com exceção das instâncias com 500 itens, nas quais o algoritmo genético de Chu e Beasley foi capaz de obter *gaps* menores. Ainda, na configuração com 30 recursos e 500 itens, o algoritmo HDP+LBC também obteve soluções suavemente melhores. Precisa-se ressaltar, no entanto, que para alcançar soluções com essa qualidade, o algoritmo de Chu e Beasley demandou tempo de execução muito maior, enquanto o HDP+LBC executou por aproximadamente o mesmo tempo nas instâncias de 30 recursos. Os tempos de execução do algoritmo NP(R) não puderam ser comparados porque não estavam disponíveis no trabalho de referência.

Os melhores resultados obtidos para as instâncias de GK foram comparados aos alcançados pelas duas configurações da abordagem híbrida baseada no algoritmo de distribuição de estimativas de Wang, Wang e Xu (HEDA1 e HEDA2) [74], pela abor-

dagem híbrida de programação linear e busca tabu de Vasquez e Hao (TS^{MKP}) [71], pelo algoritmo híbrido baseado em busca de harmonias (HHS) [78] e pela abordagem heurística usando matriz de intercepção de Veni e Balachandar (DPHEU) [72]. A Tabela 8.3 mostra os valores das soluções obtidas pelos algoritmos comparados e pelo CPLEX (valores apresentados em [72]), usado como referencial.

Tabela 8.3: Comparação do resultados para as instâncias de GK.

m	n	Melhor	HEDA1	HEDA2	TS^{MKP}	HHS	DPHEU	CPLEX
15	100	3756,97	3754,05	3737,75	3766	3756,45	3766	3766
25	100	3933,60	3950,15	3919,25	3958	3951,65	3958	3958
25	150	5614,13	5638,60	5584,85	5656	5646,25	5656	5650
50	150	5735,50	5744,52	5690,95	5767	5757,20	5767	5764
25	200	7543,67	7536,05	7443,95	7560	7552,25	7557	7557
50	200	7611,40	7637,95	7561,25	7677	7660,30	7672	7672
25	500	19165,40	19076,59	18860,29	19220	19201,30	19215	19215
50	500	18748,90	18656,61	18476,49	18806	18775,85	18806	18801
25	1500	58016,23	–	56606,10	58087	57999,35	58085	58085
50	1500	57119,47	–	56079,13	57295	57149,45	57294	57292
100	2500	95018,17	–	93578,93	95237	94844	95231	95231

Não foi possível comparar os tempos de execução, pois os trabalhos de referência não os apresentam para as instâncias maiores presentes no conjunto GK. Entretanto, Vasquez e Hao [71] comentam que sua abordagem precisou de até três dias para alcançar a solução nas instâncias com mais de 1000 itens. Similarmente, Wang, Wang e Xu [74] afirmam que HEDA1 não foi capaz de resolver as três maiores instâncias, visto que o operador de reparação depende da solução do problema de relaxamento de programação linear associado à mochila e tal solução é muito custosa quando o total de itens e recursos é muito grande. A abordagem de Vasquez e Hao alcançou as melhores soluções em todas as instâncias, mas os resultados da implementação sequencial de GRASP com *path-relinking* tiveram qualidade no máximo 1,21% inferior aos obtidos pelos autores na instância com 50 recursos e 1500 itens, mas deve-se salientar que para alcançar tal resultado, foi necessária menos de uma hora de execução (2646,9373 segundos), enquanto o algoritmo de Vasquez e Hao precisou de três dias. A implementação equivalente usando GPGPU teve qualidade, no máximo, 1,35% inferior na mesma instância, mas com aproximadamente trinta minutos de execução (1834,6083 segundos).

CAPÍTULO 9

Conclusões

Um dos problemas mais conhecidos da área de otimização combinatória, o problema da mochila multidimensional possui diversas aplicações práticas e tem sido vastamente estudado nas últimas décadas. Apesar do amplo número de pesquisas em torno do tema e das diversas alternativas propostas para resolver o problema, nenhuma delas se mostrou eficaz em obter soluções exatas para todas as instâncias em tempo viável, de fato, o problema da mochila multidimensional é comprovadamente $\mathcal{NP}$-difícil.

Como forma de obter soluções alternativas de boa qualidade para o problema da mochila multidimensional, um amplo número de técnicas tem sido utilizado. Entre as abordagens mais comuns estão os algoritmos de aproximação e as heurísticas. Os algoritmos de aproximação são capazes de produzir soluções cujas qualidades respeitam um limite máximo de distância com relação à solução ótima. Diferentemente, as heurísticas não possuem garantia alguma com relação à qualidade das soluções produzidas mas, na prática, têm se mostrado capazes de obter soluções de boa qualidade para problemas $\mathcal{NP}$-difíceis.

Ainda que possibilitem a obtenção de soluções para problemas cuja solução exata não é viável, as heurísticas podem demandar bastante tempo para chegar a resultados de qualidade satisfatória. Essa situação motiva a busca por alternativas para reduzir o tempo de execução de heurísticas, entre as quais destacam-se as estratégias de paralelização, que têm se mostrado capazes não apenas de diminuir o tempo necessário para obter boas soluções, como também de levar a soluções de qualidade ainda melhores que as implementações sequenciais.

Este trabalho teve como foco o estudo de um conjunto de metaheurísticas para resolver o problema da mochila multidimensional, a saber algoritmos genéticos, redes neurais aumentadas, GRASP e *simulated annealing*. Com base nos estudos realizados, propôs-se uma nova versão de rede neural aumentada hibridizada com características de GRASP, a qual obteve resultados relevantes. Para cada uma das metaheurísticas foram implementadas versões sequenciais e paralelas e as qualidades das soluções alcançadas e os tempos de execução foram comparados entre si e com trabalhos de referência encontrados na bibliografia pesquisada. Para possibilitar a comparação com outros trabalhos, três diferentes conjuntos de testes foram usados: o conjunto SAC-94, a biblioteca ORLIB e o conjunto GK.

As versões paralelas das metaheurísticas foram implementadas usando GPGPU devido à elevada capacidade de processamento fornecida pelas placas de vídeo modernas e à relativa facilidade de acesso a elas, quando comparado ao uso de máquinas paralelas ou *clusters*. O uso de GPGPU se mostrou eficaz, levando, em diversas das implementações propostas, a reduções nos tempos de execução e/ou soluções de qualidade superior.

Os algoritmos genéticos foram a primeira metaheurística pesquisada e, assim como as redes neurais aumentadas, também foram implementados usando uma abordagem paralela com troca de mensagens, proporcionada pelo uso da biblioteca MPI. A comparação das versões paralelas mostrou que o uso de MPI não foi tão bem sucedido como o de GPGPU, o que fez com que tal abordagem fosse abandonada nas implementações das metaheurísticas subsequentes.

Como os tamanhos das instâncias de cada conjunto diferem bastante, o comportamento das metaheurísticas foi significativamente diferente para cada um deles. Para as instâncias de tamanho reduzido de SAC-94, pôde-se observar que a implementação sequencial de redes neurais aumentadas foi capaz de obter os melhores resultados. Presume-se que essa vantagem se deva ao fato de que a capacidade de aprendizado, em conjunto com as características gulosas dessa técnica, faz com que espaços de soluções melhores sejam explorados de maneira menos errática, quando comparada a alternativas que tentam uma grande ampliação do espaço de busca para fugir de máximos locais de baixa qualidade.

Com as instâncias de ORLIB, as diferentes variantes de GRASP foram as que levaram às soluções de melhor qualidade, em especial, a versão sequencial com o uso da intensificação com *path-relinking*. Essa situação mostra que a introdução de um nível controlado de aleatoriedade, proporcionado pela construção da solução inicial com o uso de uma lista restrita de candidatos, faz com que o espaço de buscas seja ampliado por um número maior de vizinhanças. Além disso, o uso da estratégia de intensificação permite que características de soluções de elite, encontradas ao longo do processo, sejam associadas à solução.

Três abordagens se revezaram na obtenção dos melhores resultados para as instâncias do conjunto GK: implementação sequencial de GRASP com *path-relinking*, implementação paralela de redes neurais aumentadas e implementação paralela de *simulated annealing*. Apesar do maior número de melhores soluções alcançadas por GRASP com *path-relinking*, seguido pelas redes neurais aumentadas, as soluções alcançadas por *simulated annealing* foram, na média, melhores que as demais, demandando menos tempo de execução do que as demais metaheurísticas em 8 das 11 instâncias do conjunto.

Pela análise dos resultados obtidos, pode-se concluir que as metaheurísticas são alternativas bastante interessantes para resolver problemas $\mathcal{NP}$-difíceis. As soluções possibilitadas pelo uso de tais técnicas são de boa qualidade e podem ser obtidas em tempo viável. Adicionalmente, o uso de estratégias de paralelização associadas às metaheurísticas implementadas se mostrou eficaz, com capacidade de reduzir tempos de execução e, em alguns casos, melhorar a qualidade das soluções.

Como trabalhos futuros, sugere-se estudar variantes às etapas de algumas das metaheurísticas como, por exemplo, as fases de busca local e de *path-relinking* no GRASP. A associação de outras características de GRASP à rede neural aumentada é outra possibilidade que deve ser explorada, bem como o estudo de outras alternativas para

hibridização das técnicas estudadas. Outra vertente que merece especial atenção é a experimentação com diferentes estratégias de paralelização, através da exploração mais detalhada dos recursos fornecidos pelas GPUs, com a utilização de outras formas de mapeamento das estruturas para as diferentes memórias e melhor adequação do padrão de acesso, de forma a acelerar a execução dos programas usando GPGPU.

Referências Bibliográficas

[1] *GAlib - A C++ Library of Genetic Algorithm Components*. Jun. de 2015. URL: http://lancet.mit.edu/ga/.

[2] *OOMPI - Object Oriented Message Passing Interface*. Jun. de 2015. URL: http://www.osl.iu.edu/research/oompi/.

[3] A. Agarwal, H. Pirkul e V. Jacob. "Augmented Neural Networks for Task Scheduling". Em: *European Journal of Operational Research* 151.3 (2003), pp. 481–502.

[4] A. Agarwal, V.S. Jacob e H. Pirkul. "An Improved Augmented Neural-Networks Approach for Scheduling Problems". Em: *INFORMS Journal on Computing* 18.1 (2006), pp. 119–128.

[5] R.M. Aiex, M.G.C. Resende, P.M. Pardalos e G. Toraldo. *GRASP with path-relinking for the three-index assignment problem*. Relatório Técnico. AT&T Labs Research, 2000.

[6] R. Almeida e M.T.A. Steiner. "Aplicação de uma Rede Neural Aumentada para Resolução de Problemas de Corte e Empacotamento Utilizando Novas Estratégias de Aprendizagem". Em: *Simpósio Brasileiro de Pesquisa Operacional* (2013).

[7] A. Arin e G. Rabadi. "Meta-RaPS with Path Relinking for the 0-1 multidimensional knapsack problem". Em: *Intelligent Systems (IS), 2012 6th IEEE International Conference*. Set. de 2012, pp. 347–352. DOI: 10.1109/IS.2012.6335159.

[8] J.E. Beasley. "OR-Library: distributing test problems by electronic mail". Em: *Journal of the Operational Research Society* 41.11 (1990), pp. 1069–1072.

[9] D. Bertsimas e J. Tsitsiklis. "Simulated Annealing". Em: *Statistical Science* 8.1 (fev. de 1993), pp. 10–15. DOI: 10.1214/ss/1177011077.

[10] C. Blum e A. Roli. "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison". Em: *ACM Computing Surveys* 35.3 (2003), pp. 268–308. DOI: 10.1145/937503.937505.

[11] V. Boyer, M. Elkihel e D. El Baz. "Heuristics for the 0-1 multidimensional knapsack problem". Em: *European Journal of Operational Research* 199.3 (2009), pp. 658–664.

[12] R.E. Burkard, E. Çela, S.E. Karisch e F. Rendl. *QAPLIB - A Quadratic Assignment Problem Library*. Jun. de 2015. URL: http://anjos.mgi.polymtl.ca/qaplib/.

[13] E.N. Cáceres e C. Nishibe. "0-1 Knapsack Problem: BSP/CGM Algorithm and Implementation". Em: *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005)*. 2005, pp. 331–335.

[14] E. Cantú-Paz. "A Survey of Parallel Genetic Algorithms". Em: *Calculateurs Paralleles* 10 (1998).

[15] A. Caprara, H. Kellerer, U. Pferschy e D. Pisinger. "Approximation algorithms for knapsack problems with cardinality constraints". Em: *European Journal of Operational Research* 123.2 (2000), pp. 333–345. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(99)00261-1.

[16] V. Cerny. "A thermodynamic approach to the traveling salesman problem: An efficient simulation". Em: *Journal of Optimization Theory and Applications* 45 (1985), pp. 41–51.

[17] P.C. Chu e J.E. Beasley. "A Genetic Algorithm for the Multidimensional Knapsack Problem". Em: *Journal of Heuristics* 4.1 (jun. de 1998), pp. 63–86. ISSN: 1381-1231. DOI: 10.1023/A:1009642405419.

[18] B.A. Dantas e E.N. Cáceres. "A Parallel Implementation to the Multidimensional Knapsack Problem Using Augmented Neural Networks". Em: *Proceedings of the 2014 Latin American Computing Conference (CLEI)*. 2014, pp. 570–578.

[19] B.A. Dantas e E.N. Cáceres. "Implementações Paralelas para o Problema da Mochila Multidimensional usando Algoritmos Genéticos". Em: *Anais do XLVI Simpósio Brasileiro de Pesquisa Operacional (SBPO)*. 2014, pp. 1984–1994.

[20] B.A. Dantas e E.N. Cáceres. "Sequential and Parallel Implementation of GRASP for the 0-1 Multidimensional Knapsack Problem". Em: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015 Computational Science at the Gates of Nature, pp. 2739–2743. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.411.

[21] B.A Dantas e E.N. Cáceres. "Sequential and Parallel Hybrid Approaches of Augmented Neural Networks and GRASP for the 0-1 Multidimensional Knapsack Problem". Em: *The 16th International Conference on Computational Science and Its Applications (ICCSA 2016): , Beijing, China, July 4-7, 2016, Proceedings, Part II*. Ed. por O. Gervasi, B. Murgante, S. Misra, A.M.C Rocha, C. Torre, D. Taniar, B.O. Apduhan, E. Stankova e S. Wang. Springer International Publishing, 2016, pp. 207–222. ISBN: 978-3-319-42108-7. DOI: 10.1007/978-3-319-42108-7_16.

[22] J. Deane e A. Agarwal. *Neural Metaheuristics for the Multidimensional Knapsack Problem*. Relatório Técnico. 2012.

[23] J. Deane e A. Agarwal. "Neural, Genetic, and Neurogenetic Approaches for Solving the 0-1 Multidimensional Knapsack Problem". Em: *International Journal of Management & Information Systems - First Quarter 2013* 17.1 (2013), pp. 43–54.

[24] P. Feofiloff. *MAC5727 – Algoritmos de Aproximação*. Último acesso: 12/07/2016. 2003. URL: http://www.ime.usp.br/~pf/mac5727-2003/.

[25] A.M. Ferreiro, J.A. García, J.G. López-Salas e C. Vásquez. "An efficient implementation of parallel simulated annealing algorithm in GPUs". Em: *Journal of Global Optimization* 57.3 (2013), pp. 863–890.

[26] P. Festa e M.G.C. Resende. *GRASP: An Annotated Bibliography*. Relatório Técnico. AT&T Labs Research, 2001.

[27] P. Festa e M.G.C. Resende. "An annotated bibliography of GRASP – Part II: Applications". Em: *International Transactions in Operational Research* 16.2 (2009), pp. 131–172. ISSN: 1475-3995. DOI: 10.1111/j.1475-3995.2009.00664.x.

[28] H. Fingler, E.N. Cáceres, H. Mongelli e S.W. Song. "A CUDA based Solution to the Multidimensional Knapsack Problem Using the Ant Colony Optimization". Em: *Procedia Computer Science* 29 (2014). 2014 International Conference on Computational Science, pp. 84–94. ISSN: 1877–0509. DOI: 10.1016/j.procs.2014.05.008.

[29] H. Fingler. "Otimização de Colônias de Formigas em CUDA: O Problema da Mochila Multidimensional e o Problema Quadrático de Alocação". Diss. de mestrado. Faculdade de Computação - Universidade Federal de Mato Grosso do Sul, 2013.

[30] A.M. Frieze e M. Clarke. "Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst-case and probabilistic analyses". Em: *European Journal of Operational Research* 15 (1984), pp. 100–109.

[31] P.C. Gilmore e R.E Gomory. "The theory and computation of knapsack functions". Em: *Operations Research* 14.6 (1966), pp. 1045–1074.

[32] F. Glover e G. Kochenberger. *Benchmarks for the multiple knapsack problem*. abr de 2016. URL: http://hces.bus.olemiss.edu/tools.html.

[33] F. Glover. "Tabu search and adaptive memory programing – Advances, applications and challenges". Em: *Interfaces in Computer Science and Operations Research*. Kluwer, 1996, pp. 1–75.

[34] R.L. Haupt e S.E. Haupt. *Practical Genetic Algorithms*. 2ª ed. Wiley-Interscience, 2004.

[35] R.R. Hill, Y.K. Cho e J.T. Moore. "Problem reduction heuristic for the 0–1 multidimensional knapsack problem". Em: *Computers & Operations Research* 39.1 (2012). Special Issue on Knapsack Problems and Applications, pp. 19–26. ISSN: 0305-0548. DOI: 10.1016/j.cor.2010.06.009.

[36] A. Hoff, A. Løkkentangen e I. Mittet. "Genetic algorithms for 0/1 multidimensional knapsack problems". Em: *Proceedings Norsk Informatikk Konferanse*. 1996, pp. 291–301.

[37] J.H. Holland e S.E. Haupt. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[38] O.H. Ibarra e C.E. Kim. "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems". Em: *Journal of the ACM* 22.4 (1975), pp. 463–468. ISSN: 0004-5411. DOI: 10.1145/321906.321909.

[39] H. Kellerer e U. Pferschy. "A new fully polynomial approximation scheme for the knapsack problem". Em: *Approximation Algorithms for Combinatiorial Optimization: International Workshop APPROX'98 Aalborg, Denmark, July 18–19, 1998 Proceedings*. Ed. por K. Jansen e J. Rolim. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 123–134. ISBN: 978-3-540-69067-2. DOI: 10.1007/BFb0053969.

[40] S. Khuri, T. Bäck e J. Heitkötter. "The zero/one multiple knapsack problem and genetic algorithms". Em: *Proceedings of the 1994 ACM symposium on Applied computing*. 1994, pp. 188–193.

[41] S. Kirkpatrick, C.D. Gelatt e M.P. Vecchi. "Optimization by simulated annealing". Em: *Science (New York, N.Y.)* 220.4598 (maio de 1983), pp. 671–680. ISSN: 0036-8075. DOI: 10.1126/science.220.4598.671.

[42] G. Kochenberger, B. McCarl e F. Wyman. "A Heuristic for General Integer Programming". Em: *Decision Sciences* 5 (1974), pp. 36–44.

[43] M. Laguna e R. Martí. "GRASP and path relinking for 2-layer straight line crossing minimization". Em: *INFORMS Journal on Computing* 11 (1999), pp. 44–52.

[44] E.L. Lawler. "Fast Approximation Algorithms for Knapsack Problems". Em: *Mathematics of Operations Research* 4.4 (1979), pp. 339–356. DOI: 10.1287/moor.4.4.339.

[45] Y. Li, P.M. Pardalos e M.G.C. Resende. "A Greedy Randomized Adaptive Search Procedure For The Quadratic Assignment Problem". Em: *Quadratic assignment and related problems*. Ed. por P.M. Pardalos e H. Wolkowicz. Vol. 16. DIMACS Series on Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1994, pp. 237–261.

[46] R. Linden. *Algoritmos Genéticos*. 3ª ed. Editora Ciência Moderna, 2012.

[47] R.T. Liu e X.J. Lv. "MapReduce-Based Ant Colony Optimization Algorithm for Multi-Dimensional Knapsack Problem". Em: *Applied Mechanics and Materials* 380–384 (2013), pp. 1877–1880.

[48] M.J. Magazine e O. Oguz. "A fully polynomial approximation algorithm for the 0–1 knapsack problem". Em: *European Journal of Operational Research* 8.3 (1981), pp. 270–273. ISSN: 0377-2217. DOI: 10.1016/0377-2217(81)90175-2.

[49] S.L. Martins, C.C. Ribeiro e M.C. Souza. "A Parallel GRASP for the Steiner Problem in Graphs". Em: *Solving Irregularly Structured Problems in Parallel*. Ed. por A. Ferreira, J. Rolim e H. Simon e S. Teng. Vol. 1457. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 285–297. ISBN: 978-3-540-64809-3. DOI: 10.1007/BFb0018547.

[50] R. Martí. "Multi-Start Methods". English. Em: *Handbook of Metaheuristics*. Ed. por F. Glover e G.A Kochenberger. Vol. 57. International Series in Operations Research & Management Science. Springer US, 2003, pp. 355–368. ISBN: 978-1-4020-7263-5. DOI: 10.1007/0-306-48056-5_12.

[51] W.S. McCulloch e W. Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". Em: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–137.

[52] M. Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.

[53] M.C.V. Nascimento, M.G.C. Resende e F.M.B. Toledo. "GRASP heuristic with path-relinking for the multi-plant capacitated lot sizing problem". Em: *European Journal of Operational Research* 200.3 (2010), pp. 747–754.

[54] S. Niar e A. Freville. "A Parallel Tabu Search Algorithm for The 0-1 Multidimensional Knapsack Problem". Em: *Proceedings of the 11th International Symposium on Parallel Processing (IPPS'97)*. 1997, pp. 512–516.

[55] E. Onbaşoğlu e L. Özdamar. "Parallel Simulated Annealing Algorithms in Global Optimization". Em: *Journal of Global Optimization* 19.1 (), pp. 27–50. ISSN: 1573-2916. DOI: 10.1023/A:1008350810199.

[56] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[57] H. Pirkul. "A heuristic solution procedure for the multiconstraint zero-one knapsack problem". Em: *Naval Research Logistics (NRL)* 34.2 (1987), pp. 161–172. ISSN: 1520-6750. DOI: 10.1002/1520-6750(198704)34:2<161::AID-NAV3220340203>3.0.CO;2-A.

[58] C.B. Posadas, D.V. Ayala, J.S. Garcia, S.L. Silverio e M.T. Vidal. "A Solution to Multidimensional Knapsack Problem Using a Parallel Genetic Algorithm". Em: *International Journal of Intelligent Information Processing* 1.2 (2010), pp. 47–54.

[59] F. Qian e R. Ding. "Simulated annealing for the 0/1 multidimensional knapsack problem". Em: *Numerical Mathematics* 16 (2007), pp. 320–327.

[60] M.G.C. Resende e C.C. Ribeiro. "Greedy randomized adaptive search procedures". English. Em: *Handbook of Metaheuristics*. Ed. por F. Glover e G.A Kochenberger. Kluwer, 2002, pp. 219–249.

[61] M.G.C. Resende e C.C. Ribeiro. *GRASP with path-relinking: recent advances and applications*. Relatório Técnico TD-5TU726. AT&T Labs Research, 2003.

[62] M.G.C Resende e R.F. Werneck. "A Hybrid Heuristic for the p-Median Problem". Em: *Journal of Heuristics* 10.1 (2004), pp. 59–88. ISSN: 1572-9397. DOI: 10.1023/B:HEUR.0000019986.96257.50.

[63] M.G.C. Resende e C.C. Ribeiro. "A GRASP For Graph Planarization". Em: *NETWORKS* 29 (1997), pp. 173–189.

[64] C.C. Ribeiro, E. Uchoa e R.F. Werneck. "A Hybrid GRASP with Perturbations for the Steiner Problem in Graphs". Em: *INFORMS Journal on Computing* 14.3 (2002), pp. 228–246. ISSN: 1526-5528. DOI: 10.1287/ijoc.14.3.228.116.

[65] C.C. Ribeiro e I. Rosseti. "Efficient parallel cooperative implementations of GRASP heuristics". Em: *Parallel Computing* 33.1 (2007), pp. 21–35. ISSN: 0167–8191. DOI: 10.1016/j.parco.2006.11.007.

[66] S. Russel e P. Norvig. *Artificial Intelligence - A Modern Approach*. 3ª ed. Prentice Hall, 2010.

[67] S. Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem". Em: *Journal of the ACM* 22.1 (1975), pp. 115–124. ISSN: 0004-5411. DOI: 10.1145/321864.321873.

[68] S. Senju e Y. Toyoda. "An Approach to Linear Programming with 0-1 Variables". Em: *Management Science* 15 (1968), B196–B207.

[69] C. Solnon e D. Bridge. "An Ant Colony Optimization Meta-heuristic for Subset Selection Problems". Em: *Systems Engineering Using Swarm Particle Optimization*. Ed. por N. Nedjah e L. M. Mourelle. Nova Science Publishers, 2006, pp. 7–29.

[70] M. Varnamkhasti. "Overview of the algorithms for solving the multidimensional knapsack problems". Em: *Advanced Studies in Biology* 4.1 (2012), pp. 37–47.

[71] M. Vasquez e J.-k. Hao. "A hybrid approach for the 0–1 multidimensional knapsack problem". Em: *Proceedings of the International Joint Conference on Artificial Intelligence 2001*. 2001, pp. 328–333.

[72] K.K. Veni e S.R. Balachandar. "A new heuristic approach for large size zero–one multi knapsack problem using intercept matrix". Em: *International Journal of Computational and Mathematical Sciences* 4.5 (2010), pp. 259–263.

[73] D.S. Vianna e M.F.D. Vianna. "Local search-based heuristics for the multiobjective multidimensional knapsack problem". Em: *Produção*. Vol. 23. 3. 2013, pp. 478–487. DOI: 10.1590/S0103-65132012005000081.

[74] L. Wang, S.-y. Wang e Y. Xu. "An effective hybrid EDA-based algorithm for solving multidimensional knapsack problem". Em: *Expert Systems with Applications* 39.5 (2012), pp. 5593–5599. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2011.11.058.

[75] C. Zaiontz. *Real Statistics using Excel – Wilcoxon Rank Sum Table*. Jun. de 2016. URL: http://www.real-statistics.com/wp-content/uploads/2012/12/image3727.png.

[76] C. Zaiontz. *Real Statistics using Excel – Wilcoxon Rank Sum Test for Independent Samples*. Mar. de 2016. URL: http://www.real-statistics.com/non-parametric-tests/wilcoxon-rank-sum-test/.

[77] M. Zbierski. "A Simulated Annealing Algorithm for GPU Clusters". Em: *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*. PPAM'11. Torun, Poland: Springer-Verlag, 2012, pp. 750–759. ISBN: 978-3-642-31463-6. DOI: 10.1007/978-3-642-31464-3_76.

[78] B. Zhang, Q.-K. Pan, X.-L. Zhang e P.-Y. Duan. "An effective hybrid harmony search-based algorithm for solving multidimensional knapsack problems". Em: *Applied Soft Computing* 29 (2015), pp. 288–297. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2015.01.022.

Anexo A

Artigos Decorrentes desta Tese

Os estudos realizados ao longo do desenvolvimento deste trabalho levaram à elaboração dos seguintes artigos:

- 2014:
  - A parallel implementation to the multidimensional knapsack problem using augmented neural networks. 2014 XL Latin American Computing Conference (CLEI), Montevideo, pp. 570–578, 2014 [19].
  - Implementações Paralelas para o Problema da Mochila Multidimensional usando Algoritmos Genéticos. XLVI Simpósio Brasileiro de Pesquisa Operacional (SBPO), Salvador, pp. 1984–1994, 2014 [18].
- 2015:
  - Sequential and Parallel Implementation of GRASP for the 0-1 Multidimensional Knapsack Problem. International Conference on Computational Science (ICCS 2015) – Computational Science at the Gates of Nature, Reykjavyk. Procedia Computer Science, v. 51, pp. 2739–2743, 2015 [20].
- 2016:
  - Sequential and Parallel Hybrid Approaches of Augmented Neural Networks and GRASP for the 0-1 Multidimensional Knapsack Problem. The 16th International Conference on Computational Science and Its Applications, Pequim. ICCSA 2016 , Part II, LNCS 9787, pp. 207–222, 2016 [21].
  - A Parallelization of a Simulated Annealing Approach for 0-1 Multidimensional Knapsack Problem using GPGPU. The 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2016), Los Angeles. Aceito para publicação.

Anexo B

Tabela de Soma de *Ranks* de Wilcoxon

Para fazer a validação dos resultados alcançados pelas metaheurísticas implementadas neste trabalho usou-se o teste de somas de *ranks* de Wilcoxon. A tabela de soma de *ranks* de Wilcoxon considerando o nível de significância $\alpha = 0{,}005$ e duas caudas, necessária para aplicação do teste, é apresentada na Figura B.1 [75].

n_2\n_1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1																									
2																									
3																									
4			--	10																					
5			6	11	17																				
6			7	12	18	26																			
7		--	7	13	20	27	36																		
8		3	8	14	21	29	38	49																	
9		3	8	14	22	31	40	51	62																
10		3	9	15	23	32	42	53	65	78															
11		3	9	16	24	34	44	55	68	81	96														
12		4	10	17	26	35	46	58	71	84	99	115													
13		4	10	18	27	37	48	60	73	88	103	119	136												
14		4	11	19	28	38	50	62	76	91	106	123	141	160											
15		4	11	20	29	40	52	65	79	94	110	127	145	164	184										
16		4	12	21	30	42	54	67	82	97	113	131	150	169	190	211									
17		5	12	21	32	43	56	70	84	100	117	135	154	174	195	217	240								
18		5	13	22	33	45	58	72	87	103	121	139	158	179	200	222	246	270							
19		5	13	23	34	46	60	74	90	107	124	143	163	183	205	228	252	277	303						
20		5	14	24	35	48	62	77	93	110	128	147	167	188	210	234	258	283	309	337					
21		6	14	25	37	50	64	79	95	113	131	151	171	193	216	239	264	290	316	344	373				
22		6	15	26	38	51	66	81	98	116	135	155	176	198	221	245	270	296	323	351	381	411			
23		6	15	27	39	53	68	84	101	119	139	159	180	203	226	251	276	303	330	359	388	419	451		
24		6	16	27	40	54	70	86	104	122	142	163	185	207	231	256	282	309	337	366	396	427	459	492	
25	--	6	16	28	42	56	72	89	107	126	146	167	189	212	237	262	288	316	344	373	404	435	468	501	536

Figura B.1: Tabela de soma de *ranks* de Wilcoxon, $\alpha = 0{,}05$, duas caudas.