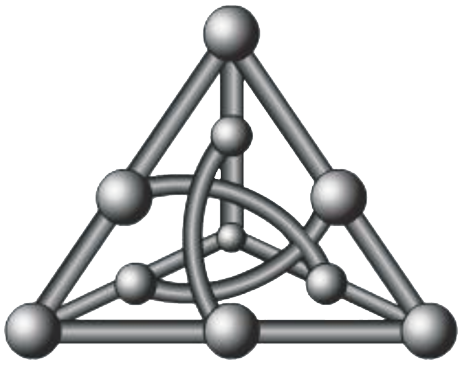
# OTIMIZAÇÃO DE COLÔNIAS DE FORMIGAS EM CUDA: O PROBLEMA DA MOCHILA E O PROBLEMA QUADRÁTICO DE ALOCAÇÃO

HENRIQUE FINGLER {HENRIQUE.FINGLER@GMAIL.COM}



Faculdade de Computação (FACOM)
Universidade Federal do Mato Grosso do Sul (UFMS)

Orientador: Henrique Mongelli

Abril de 2013 – Versão Final

# RESUMO

O problema da mochila binária multidimensional e o problema quadrático de alocação não possuem algoritmos exatos que encontrem a solução ótima em tempo viável para uma aplicação prática, a não ser que as instâncias sejam muito pequenas. Nestes casos, algoritmos aproximados, que não garantem encontrar a solução ótima, mas tentam encontrar uma próxima, podem ser utilizados. Dentre estes algoritmos, temos as meta-heurísticas, que são algoritmos não-dependentes do problema que iterativamente buscam no espaço de soluções uma solução satisfatória. Uma destas meta-heurísticas é a otimização de colônia de formigas, que imita o comportamento de formigas procurando por comida. Quando estas formigas caminham pelo território, depositam feromônios, que é o método de comunicação entre elas e as guiam a fontes de comida já encontradas. Devido ao poder de computação das placas de vídeo atuais, a implementação foi feita em paralelo usando CUDA, a linguagem para *GPGPUs* da NVIDIA.

palavras-chave: meta-heurística, CUDA, problema da mochila, problema quadrático de alocação.

# SUMÁRIO

# LISTA DE FIGURAS

## LISTA DE TABELAS

# 1 INTRODUÇÃO

Encontrar as soluções ótimas de problemas de otimização, que podem possuir aplicações industriais ou científicas, pode ser um problema intratável para alguns casos. Pode ser que instâncias muito grandes devam ser solucionadas em um pequeno espaço de tempo, ou deseja-se encontrar as soluções ótimas de problemas complexos. No caso onde o tempo para encontrar uma solução é o fator determinante na resolução de uma instância de um problema, uma solução próxima da ótima talvez seja suficiente.

A solução ótima pode ser formulada por meio de um problema de otimização combinatória onde cada possível solução tem um valor numérico e deseja-se encontrar a melhor delas, a menor se o problema for de minimização ou a maior se o problema for de maximização. Se as variáveis deste problema forem discretas, o problema é chamado de problema de otimização combinatória inteira.

Em várias situações, o problema pertence à classe dos problemas NP-difícil [5], ou seja, não existe um algoritmo de complexidade polinomial para solucioná-lo, a não ser que $P = NP$. Se o tamanho da instância do problema a ser resolvido, mesmo sendo NP-difícil, não for muito grande, um método exato pode ser utilizado, porém, conforme o tamanho da instância cresce, o tempo consumido para resolvê-lo aumenta a ponto de não ser razoável para uma aplicação prática. Nestes casos, encontrar a solução ótima torna-se inviável, mas boas soluções podem ser obtidas por algoritmos aproximados: de aproximação, se possível, ou métodos heurísticos. No caso de um algoritmo de aproximação, tem-se a garantia da distância máxima da solução encontrada e a ótima. Nos métodos heurísticos, ao contrário dos métodos de aproximação, não dão nenhuma garantia sobre a qualidade da solução.

Quando um algoritmo de aproximação não existe ou a qualidade da solução não é suficiente e os métodos exatos não são viáveis, um dos métodos heurísticos pode ser utilizado e obter resultados satisfatórios. Entre estes métodos, os mais comuns são as meta-heurísticas, algoritmos independentes do problema que iterativamente buscam soluções seguindo um comportamento. Algumas meta-heurísticas são baseadas em processos naturais, como colônia de abelhas e formigas, algumas tomam decisões determinísticas e tem execução igual quando a entrada é a mesma, outras são estocásticas, ou seja, em alguma parte do algoritmo uma decisão com base em variáveis aleatórias é tomada.

Entre os problemas que pertencem à classe NP-difícil, tem-se o problema da mochila binária multidimensional (MKP) e o problema quadrático de alocação (QAP). A escolha destes problemas deve-se ao fato que ambos têm diversas aplicações práticas e teóricas. Para o MKP, alguns exemplos são:

- Aparece com grande frequência como um subproblema de muitos problemas mais complexos [6, 7, 8, 9, 10];
- Surge em diversas situações práticas como na escolha de portfólios de investimento, manufatura de placas de sistemas integrados e até mesmo em criptografia [11, 10, 6].

No caso do QAP algumas das áreas onde é utilizado são:

- Organização de setores de um Hospital [12];
- Problema de cabeamento de Steinberg [13];
- Nos problemas de particionamento de grafos e clique máximo, problema do caixeiro viajante, problema do arranjo linear, problema de empacotamento em grafos e outros podem ser reduzidos ao QAP [1].

Como ambos os problemas são NP-difíceis, este trabalho tenta encontrar boas soluções utilizando a meta-heurística otimização de colônia de formigas (ACO) [14] em paralelo, cujo comportamento, que será introduzido a seguir, possui vários passos que podem ser executados concorrentemente. Este é o principal motivo da escolha deste método, pois pode ser implementado em paralelo em uma GPGPU. Uma placa de vídeo (*graphics processing unit* ou GPU) com a capacidade de executar código não exclusivamente gráfico é chamada de GPGPU (*General Purpose Graphics Processing Unit* ou Unidade de Processamento Gráfico de Propósito Geral). Além do CUDA da NVIDIA, que foi utilizado neste trabalho, as tecnologias mais utilizadas para executar em uma GPGPU são: FireStream da AMD, OpenCL que é livre e mantida por um conjunto de organizações e DirectCompute da Microsoft.

A meta-heurística de otimização colônia de formigas, ou ACO, tenta imitar o comportamento de formigas procurando por comida a partir de um formigueiro. Quanto mais perto a fonte de comida está do formigueiro, melhor. Enquanto as formigas caminham, inicialmente aleatoriamente, vão depositando um feromônio, que serve para guiá-la de volta ao formigueiro e também para que outras formigas possam seguir o caminho já realizado. Este feromônio evapora com o tempo, portanto um caminho que tem um grande fluxo de formigas terá mais feromônio que um caminho onde poucas formigas passam. Quando uma formiga encontra uma fonte de comida, ela volta ao formigueiro, se esta fonte está muito longe do formigueiro, boa parte do feromônio já evaporou, se está perto, ainda existe uma boa quantidade depositada no caminho de volta. Isto faz com que fontes próximas, devido ao menor tempo de viagem de ida e volta, tenham mais feromônio que fontes afastadas. Formigas que saem do formigueiro após as primeiras já terem saído aleatoriamente, fazem uma escolha entre os caminhos com

feromônio, quanto mais forte (mais feromônio) maior a probabilidade de a formiga seguir este caminho.

A analogia feita por este comportamento é que as fontes de comida são as soluções para o problema, quanto mais próxima do formigueiro, melhor é a solução. O feromônio depositado é como as formigas artificiais se comunicam, tentando compartilhar informações sobre onde uma boa solução pode ser encontrada. Como este feromônio é representado depende da implementação, pode ser uma tabela, um vetor ou qualquer outra estrutura de dados.

Como o trabalho feito pelas formigas é independente, é simples paralelizar um algoritmo ACO, basta separar cada formiga para um processador. Se esta organização for implementada em uma GPGPU pode-se obter um ganho significativo [4], porém através de uma organização diferente, onde cada formiga é mais do que um processador, o resultado pode ser melhorado [15]. Neste trabalho, esta segunda organização, que é mais eficiente e consome menos tempo de execução, foi implementada utilizando CUDA, uma linguagem de programação baseada em C que é executada em placas de vídeo da NVIDIA.

Os resultados obtidos mostram que a otimização de colônia de formigas é um método viável para resolver até instâncias grandes, principalmente se for implementado em paralelo.

# 2 FUNDAMENTOS

## 2.1 ALGORITMOS

A execução de um algoritmo utiliza vários recursos, que podem ser memória, comunicação ou, o que normalmente é o mais importante, tempo computacional. Pode-se fazer uma análise em um algoritmo para prever a quantidade de cada recurso que será utilizada em sua execução. Geralmente, ao analisar vários algoritmos candidatos que resolvem um problema, o mais eficiente - o que utiliza menos recursos - pode ser facilmente identificado. Tal análise pode indicar mais do que um candidato viável, mas vários algoritmos inferiores, que são algoritmos que obtêm pior desempenho em relação aos outros, como complexidade maior, por exemplo, são descartados no processo.

Antes de analisar um algoritmo, um modelo de tecnologia de implementação que será usado deve ser escolhido e deve incluir um modelo para os recursos daquela tecnologia e seus custos. Normalmente o modelo utilizado para esta análise é a Máquina de Turing [16].

Utilizando o modelo definido, pode-se contar o número de instruções ou passos que o algoritmo executa em relação às entradas do algoritmo em três situações: melhor caso, pior caso e caso médio. Com algumas abstrações nesta análise pode-se definir a taxa ou ordem de crescimento do algoritmo. Considera-se que um algoritmo é melhor que outro quando, no pior caso, sua taxa de crescimento é menor. Devido a fatores constantes e termos de ordem baixa, esta comparação pode ser errônea quando a entrada é pequena.

Esta taxa de crescimento de um algoritmo é uma característica simples para definir a eficiência de um algoritmo e permite comparações relativas. Quando o tamanho da entrada dos algoritmos é grande o suficiente, e tende a um valor limitante muito grande a ponto de fazer com que apenas a taxa de crescimento seja relevante, esta taxa representa a eficiência assintótica do algoritmo. Normalmente um algoritmo que é assintoticamente mais eficiente é a melhor escolha para todos os tamanhos, menos os muito pequenos.

A notação mais utilizada para representar a função assintótica de um algoritmo é a “big O notation”, ou notação O grande, que determina um limite superior da função dentro de um fator constante. Detalhes sobre análise de algoritmos e notações de eficiência podem ser encontrados em [5].

Dado um problema, existe um conjunto que possui todas as possíveis soluções, e deve-se encontrar uma ou mais que o solucionem. Se este conjunto contém apenas dois elementos, que podem ser chamados de "sim" e "não", este problema é chamado de problema de decisão. Se o conjunto de soluções possui diversos elementos, e cada um destes elementos tem um valor associado, chama-se este problema de problema de otimização. Em um problema de otimização, se a solução que deve ser encontrada é a solução de valor máximo, o problema é dito de maximização. Caso contrário o problema é dito de minimização.

Pode-se representar um problema de otimização como uma função $f : A \to \mathbb{R}$, onde $A$ é um conjunto de números reais, e procura-se um $x_0$ em $A$ tal que $f(x_0) \leqslant f(x)$ para $\forall x$ se o problema for de minimização, ou $f(x_0) \geqslant f(x)$ para $\forall x$. Intuitivamente, pode-se comparar todos os valores do conjunto $A$ e encontrar o maior ou menor, porém normalmente o tamanho de $A$ é suficientemente grande para que esta busca seja incabível.

Para resolver um problema de otimização, existem dois possíveis caminhos a seguir: métodos exatos e métodos aproximados. Os métodos exatos encontram uma solução ótima e garantem sua otimalidade. Para problemas NP-completos, não existem algoritmos exatos com complexidade polinomial, a não ser que $P = NP$. Métodos aproximados podem encontrar soluções de boa qualidade em um tempo razoável para uso prático, porém não dão a garantia de que a solução ótima será encontrada. A Figura 2.2 mostra uma representação e exemplos dos métodos exatos e aproximados.

Um problema C é NP-completo, ou formalmente pertencente à classe de problemas NP-completos, se o problema está no conjunto de problemas NP ($C \in NP$) e todos os problemas pertencentes à NP são redutíveis a C. Para um problema pertencer à classe NP, este deve ser um problema de decisão e ter sua solução verificável em tempo polinomial. Um problema C é NP-difícil se todos os problemas em NP podem ser reduzidos a C, mas não é necessário que sua solução seja verificável em tempo polinomial. Mais detalhes sobre as classes e redução de problemas podem ser encontrados em [5]. A Figura 2.1 mostra os conjuntos de problemas para os casos de $P \neq NP$ e $P = NP$.

Na classe de métodos exatos encontram-se alguns algoritmos clássicos, como programação dinâmica e *branch-and-X* (*branch-and-bound*, *branch-and-cut*, *branch-and-price*). Estes métodos enumerativos podem ser vistos como algoritmos de busca em árvores. A busca é feita no espaço de soluções, e o problema resolvido através da subdivisão do espaço em subespaços menores.

A programação dinâmica é baseada na divisão recursiva do problema em subproblemas e tem como raiz o princípio de Bellman, que diz que dado uma solução ótima de um problema, estão contidas nesta solução soluções ótimas para subproblemas do problema original. Este método encontra a solução ótima através de uma sequência de decisões parciais, e evita buscar todo o espaço de soluções pois corta sequências de decisões que não levam à solução ótima.

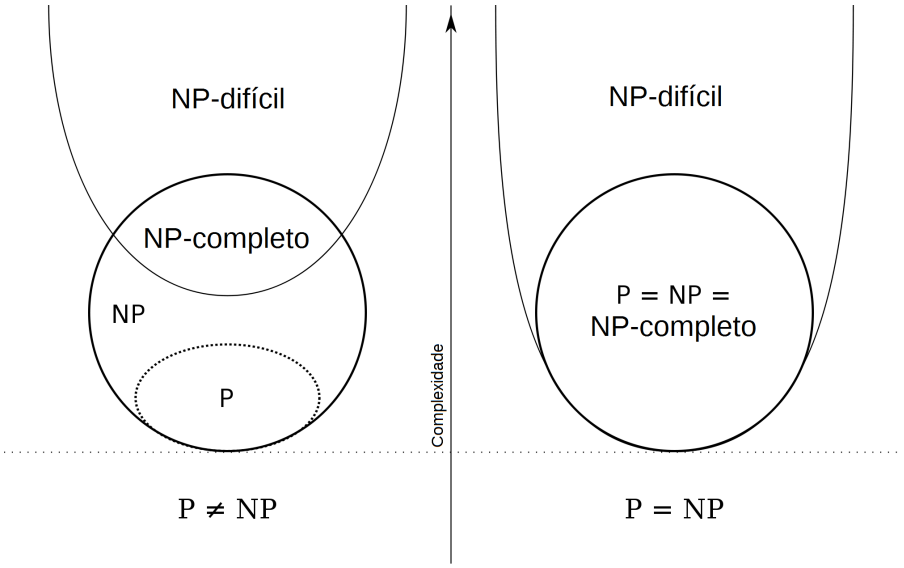



Figura 2.1: Representação dos conjuntos P, NP, NP-completo e NP-difícil.

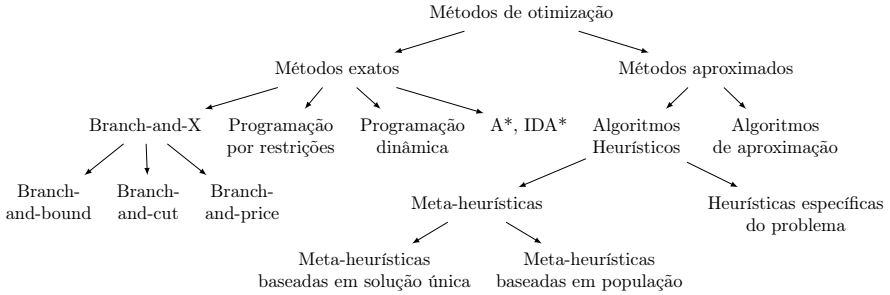



Figura 2.2: Métodos para resolver um problema. [2]

O *branch-and-X* utiliza uma enumeração implícita de todas as soluções do problema considerado. O espaço de busca é explorado através da representação de uma árvore, a raiz é o problema a ser resolvido, as folhas são as possíveis soluções e cada nó interno é um subproblema. Ao se deparar com um nó interno que não é promissor, ou seja, não contém a solução ótima, a sub-árvore com raiz neste nó interno é cortada do espaço de busca.

Ainda sobre os métodos exatos, em alguns casos, e assumindo que $P \neq NP$, a complexidade de um problema pode ser super-polinomial quando apenas o tamanho da entrada é levado em consideração, mas quando sua complexidade é expressa em relação ao tamanho da entrada e mais algum outro parâmetro k dependente do problema, o fator da complexidade relativo ao tamanho da entrada deixa de ser exponencial, e esta exponencialidade é movida ao parâmetro k. Portanto, se este parâmetro k for fixado em um valor pequeno e a taxa de crescimento da função sobre k é relativamente pequena, então tal problema pode ser considerado tratável, mesmo tradicionalmente considerado intratável por não ter solução polinomial.

Algoritmos que utilizam deste conceito de complexidade parametrizada para resolver um problema são chamados de algoritmo de parâmetro-fixo

tratável, ou FPT ("fixed-parameter tractable algorithm"). Mais detalhes sobre complexidade parametrizada e algoritmos FPT pode ser encontrado em [17].

Um algoritmo pode ser paralelizado para diminuir o seu tempo de execução. Quando paralelizado, o código é executado por mais de um processador, seja um processador com vários núcleos, uma placa de vídeo ou grandes redes de computadores trabalhando em conjunto.Com a programação paralela, pode-se utilizar métodos exatos em instâncias pequenas de problemas complexos, porém, em instâncias muito grandes, mesmo utilizando programação paralela, o tempo consumido pode ser alto demais na prática. Nestes casos um algoritmo aproximado pode obter melhores resultados.

A classe de métodos aproximados pode ser dividida em duas subclasses: algoritmos heurísticos e algoritmos de aproximação. A diferença entre as duas subclasses está na garantia da solução obtida, enquanto os de aproximação dão garantia de o quão próxima da solução ótima estará a solução encontrada, os heurísticos não dão garantia alguma. O conceito de resolver problemas utilizando heurísticas foi introduzido por Polya em 1945 [18]

Um algoritmo de aproximação é dito $\epsilon - \mathtt{aproximado}$ [19] se sua complexidade é polinomial e, para qualquer instância do problema, o valor da solução encontrada não é menor que um fator $\epsilon$ do valor da solução ótima. Para alguns problemas NP-difíceis é impossível obter um algoritmo de aproximação a não ser que $\mathsf{P} = \mathsf{NP}$. Os algoritmos de aproximação aproveitam características do problema que estão resolvendo para encontrar a solução aproximada, portanto os algoritmos de aproximação são ditos dependentes do problema. Mais detalhes sobre algoritmos de aproximação podem ser encontrados em [19, 20].

Diferente dos métodos de aproximação, as meta-heurísticas não garantem uma distância da solução ótima, mas normalmente encontram soluções bem próximas. Também não são dependentes do problema, uma mesma meta-heurística pode ser aplicada a mais de um problema.

Existem várias aplicações de meta-heuristícas para problemas reais, como [2]:

- Aerodinâmica, dinâmica de fluidos, telecomunicações, automotivos e robótica.
- Aprendizado de máquina e mineração de dados em bioinformática e biologia computacional.
- Modelagem de sistemas, simulação e identificação em química, física e biologia.
- Planejamento em problemas de roteamento, transporte, logística e outros.

Uma meta-heurística tem dois critérios para sua execução: exploração do espaço de busca, ou diversificação, que é o quanto o espaço de soluções será explorado e aprofundamento ou intensificação da busca, que é o quanto uma certa área restrita será explorada a fundo. Na diversificação, regiões

não exploradas devem ser visitadas para garantir que o espaço de busca foi igualmente explorado e que a busca não foi restrita a apenas uma região pequena, já na intensificação, regiões promissoras são exploradas mais à fundo para que uma solução parecida, porém melhor, com uma já conhecida possa ser encontrada.

Dada uma meta-heurística, pode-se classificá-la de acordo com suas características e comportamento na execução. Algumas podem ser baseadas em mecanismos da natureza, como comportamento de formigas e abelhas, ou processos físicos como o resfriamento de placas no *simulated annealing* (SA). Uma Podem ser classificadas também como determinísticas ou estocásticas. Uma meta-heurística determinística sempre terá sua execução e resultado igual sempre que a entrada do algoritmo for a mesma, já uma estocástica utiliza variáveis aleatórias e pode obter resultados diferentes mesmo se a entrada do algoritmo for a mesma. Podem também ser baseadas em população, onde a meta-heurística possui várias soluções que são modificadas a cada passo, ou ser baseada em solução única, onde existe apenas uma solução que é modificada a cada passo. Uma meta-heurística também pode ser classificada como de construção, onde as soluções são construídas do zero, ou de modificação, onde as soluções são modificadas.

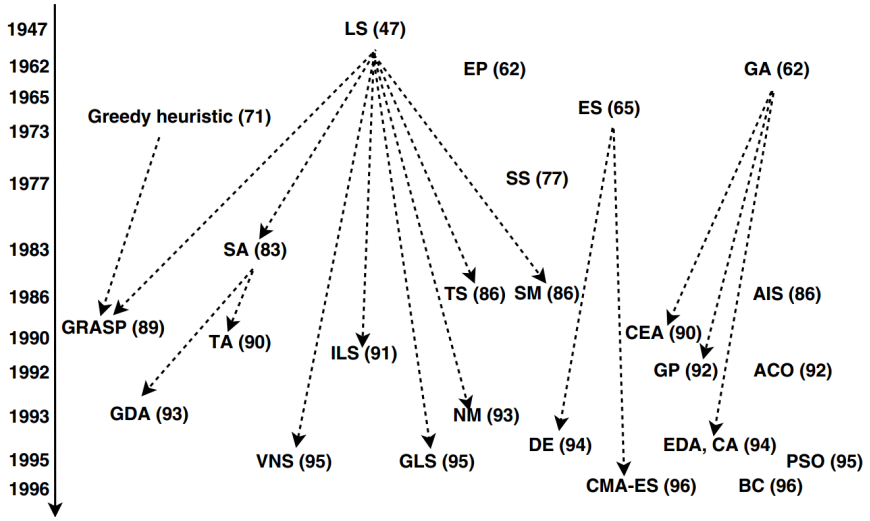



Figura 2.3: Genealogia das meta-heurísticas. [2]

Na Figura 2.3 é mostrada a genealogia das meta-heurísticas e suas siglas. Abaixo são mencionadas as siglas, nomes e referências para os trabalhos originais (estes dados foram retirados de [2]):

ACO (otimização de colônia de formigas) [14], AIS (sistemas imunológicos artificiais)[21, 22], BC (colônia de abelhas) [23, 24], CA (algoritmos culturais) [25], CEA (algoritmos coevolucionários) [26, 27], CMA-ES (estratégia de evolução adaptativa de covariância de matrizes) [28], DE (evolução diferencial) [29, 30], EDA (algoritmos de estimativa de distribuição) [31], EP (programação evolucionária) [32], ES (estratégias de evolução) [33, 34], GA (algoritmos genéticos) [35, 36], GDA (algoritmos do grande dilúvio) [37], GLS (busca lo-

cal guiada) [38, 39], GP (programação genética) [40], GRASP (procedimento de busca gulosa adaptativa) [41], ILS (busca local iterativa) [42], NM (método de ruídos) [43], PSO (otimização de enxame de partículas) [44], SA (recozimento simulado) [45, 46], SM (método de suavização) [47], SS (busca dispersa) [48], TA (aceitação de limiar) [49], TS (busca tabu) [50, 51] e VNS (busca de vizinhança variável) [52]. Maiores detalhes sobre estas heurísticas podem ser encontradas em [2].

Devido à organização e a maneira como o algoritmo de otimização de colônia de formigas é executado, e ao ganho de desempenho através uso de programação paralela, esta meta-heurística foi escolhida para ser implementada. Como será visto na seção a seguir, esta meta-heurística possui uma coleção de formigas explorando o espaço de possíveis soluções do problema e se comunicando através de feromônios. Cada formiga realiza seu trabalho independente das outras, o que faz esta meta-heurística simples de ser paralelizada, mesmo que não eficientemente.

## 2.2 OTIMIZAÇÃO DE COLÔNIA DE FORMIGAS

A meta-heurística de colônia de formigas tenta imitar o comportamento de formigas em busca de comida. Resolver um problema utilizando *ACO* (*Ant Colony Optimization*, Otimização de Colônia de Formigas) é similar a resolver o problema do melhor caminho em um grafo dirigido.

Inicialmente, as formigas andam aleatoriamente em torno do formigueiro e, durante sua caminhada, deixam uma trilha de feromônios para que consigam voltar. Quando uma fonte de comida é encontrada, esta formiga retorna ao formigueiro reforçando a trilha deixada por ela anteriormente. Outras formigas, que podem sentir estes feromônios, podem tomar decisões de seguir ou não um caminho à uma fonte de comida dependendo da intensidade. Quanto mais feromônios em um local, mais atraente ele é para as formigas. Estes feromônios evaporam com o tempo, enfraquecendo o poder de atração de todos os caminhos. Caminhos longos terão menor intensidade pois enquanto uma formiga vai até a fonte de comida e volta, boa parte dos feromônios já evaporou, enquanto que, em um caminho curto, menos feromônios terá evaporado quando a formiga retornar ao formigueiro.

Inspirado neste comportamento, Dorigo [14] propôs o algoritmo *Ant System* (*AS*) para resolver o problema do menor caminho. Os resultados iniciais desta técnica foram bons, porém não superavam outras técnicas quando o tamanho do grafo era muito grande. Entretanto, estes resultados foram importantes para que variações fossem estudadas e propostas. Dorigo definiu *ACO* como sendo um *framework* comum a todos os algoritmos que utilizam este comportamento de formigas. Portanto o *AS* é um algoritmo *ACO*.

Pode-se descrever o ACO como uma meta-heurística em que formigas artificiais se comunicam através de trilhas de feromônios, que funcionam como uma função probabilística para construir ou modificar soluções. Durante a execução do algoritmo, estas informações são adaptadas para refletir a experiência de cada formiga na busca por novas soluções. Como a informação destas trilhas de feromônio são utilizadas, atualizadas e inicializadas é a

etapa mais importante no desenvolvimento de um algoritmo de otimização de colônias de formigas.

As formigas constroem novas soluções passeando em um grafo, em que os vértices são as soluções (fontes de comida), e as arestas são caminhos (que contém ou não feromônios). Cada formiga possui dados sobre o caminho seguido e, para determinar a próxima aresta a ser visitada, utiliza estes dados em conjunto com o feromônios das arestas. Para que boas soluções sejam encontradas, várias formigas devem se comunicar indiretamente pelos feromônios, trocando informações das características de cada solução gerada.

Além do movimento concorrente das formigas, o *ACO* possui mais dois passos: evaporação de feromônios e ações globais. A evaporação dos feromônios é um processo de atualização de toda a tabela de feromônios, onde a quantidade é reduzida de uma quantia definida por uma função. Esta atualização tenta evitar a convergência de várias formigas a uma solução subótima local, fazendo com que as formigas explorem novos caminhos. As ações globais podem ser utilizadas para realizar ações centralizadas que não podem ser executadas pelas formigas isoladamente. É possível, por exemplo, iniciar um processo de busca local ou coletar informações sobre as soluções construídas por todas as formigas, para que a tabela de feromônios seja atualizada favorecendo soluções ótimas locais (esta é uma atualização que não faz parte da etapa de evaporação, apenas é feita para intensificar os feromônios em um certo local).

A Figura 2.4 mostra um pequeno exemplo de como este comportamento funciona. Inicialmente (1) uma formiga sai do formigueiro e, por qualquer caminho (a) tenta encontrar uma fonte de comida. Ao voltar ao formigueiro (b), uma trilha de feromônios é deixada por ela, para que outras possam seguir seu caminho. Devido à evaporação destes feromônios, caminhos mais curtos tendem a ter mais feromônios (2) e atrair mais formigas. Após um certo tempo (3), os caminhos utilizados pelas formigas tendem a convergir para o mais curto. Mesmo com tal convergência, algumas formigas ainda podem seguir por caminhos com poucos feromônios, devido a sua característica probabilística.

Inicialmente o *AS* era um conjunto de três algoritmos, chamados *ant-cycle*, *ant-quantity* e *ant-density*. No *ant-quantity* e no *ant-density*, os feromônios de uma aresta era atualizado logo após uma formiga andar sobre ela. Já no *ant-cycle*, as formigas primeiro andavam sobre o grafo, e atualizavam os feromônios de acordo com a qualidade da solução encontrada, assim, quanto melhor a solução, maior a quantidade de feromônios depositada. Este último algoritmo obteve melhores resultados em comparação aos outros dois, que não foram mais estudados.

Uma primeira melhoria ao algoritmo do *ant-cycle* foi introduzida por Dorigo [53, 54], chamada estratégia elitista, que consistia em aumentar a quantidade de feromônios após o passo de atualização de todas as arestas, na melhor solução global obtida até o momento. Este processo faz parte da etapa

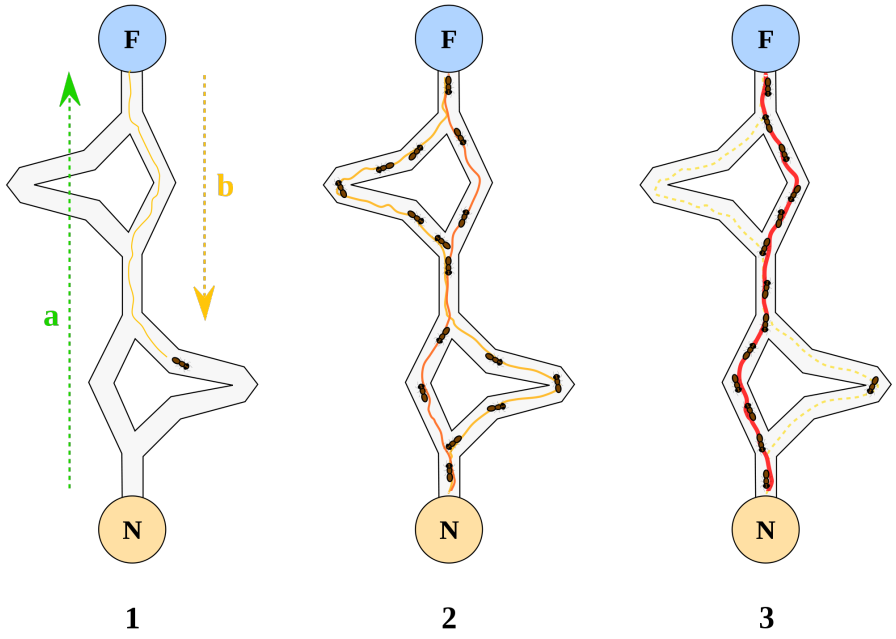



Figura 2.4: Exemplo de caminhos entre o formigueiro e uma fonte de comida. O ponto N representa o formigueiro, o F a fonte de comida. [3]

de ações globais definida anteriormente. Outras estratégias de melhoria do *ant-cycle* foram feitas, como descrito a seguir.

- $AS_{rank}$: esta melhoria se baseia na estratégia elitista, e consiste na ordenação por melhor solução das formigas e, no passo de atualização da tabela de feromônios, a formiga que obteve a melhor solução até o momento deposita feromônios utilizando uma proporção $w$, a segunda melhor $w - 1$ e assim por diante. A formiga que obteve a pior solução global não deposita nenhum feromônio.

- MIN-MAX *Ant System*: nesta variação, todas as arestas começam com uma quantidade de feromônios alta e igual, fazendo com que as formigas se espalhem pelo grafo. Assim uma grande exploração é realizada logo no começo da execução. Apenas a formiga que encontrou a melhor solução global ou local deposita feromônios. Ainda, a trilha de feromônios é controlada por valores máximos e mínimos, que são determinados pelo valor da melhor solução. Assim, nenhum valor pode passar do valor máximo, nem do valor mínimo. Se algum elemento da trilha for ultrapassar estes limites, o valor do limite é gravado.

- *Ant System Colony*: altamente elitista, esta melhoria faz com que também, apenas a formiga que obteve a melhor solução global ou local deposite feromônios. Uma grande diferença é que as formigas atualizam os feromônios enquanto geram as soluções e, ainda, enquanto caminham em uma aresta, um pouco de feromônios é "consumido", fazendo com que a probabilidade de uma formiga tomar o mesmo ca-

minho que outra seja reduzido. Ainda, uma busca local é realizada por todas as formigas antes do passo de atualização de feromônios.

Bons resultados foram obtidos quando o algoritmo de otimização de formigas foi utilizado para vários problemas. Entre eles: problema do caixeiro viajante (TSP) [55], problemas de alocação [56], problemas de classificação [57], problemas de conjuntos, como partição de conjuntos [58], problemas de roteamento de veículos [59] entre outros.

## 2.3 PROGRAMAÇÃO PARALELA

Esta área da computação tem crescido pois, é necessário cada vez mais, um maior poder de processamento para resolver instâncias de problemas muito grandes e/ou problemas muito complexos, que levariam dias, semanas ou até anos para serem resolvidos com código sequencial em um computador comum. Problemas de engenharia ou modelagem numérica, por exemplo, realizam vários cálculos repetidos em uma enorme quantidade de dados. A resposta para estes problemas normalmente é necessária em alguns segundos ou minutos, não dias. Conforme aumenta-se o volume de dados ou acrescentam-se cálculos ou funcionalidades à solução do problema para aumentar sua precisão ou aumentar o número de resultados diferentes, também cresce a quantidade de processamento necessário, por isto a computação paralela é importante. Na computação paralela, tem-se o conceito de processador com paralelismo, ou máquina paralela, que é um processador com recursos para executar diferentes porções de códigos concorrentemente, ou seja, ao mesmo tempo, ao contrário de um processador comum (sequencial), que executa apenas uma instrução em um dado instante.

Se um código escrito de forma sequencial for executado em uma máquina paralela, não haverá ganhos de desempenho, a não ser que alguma ferramenta de paralelização automatizada seja executada. Um código que foi escrito de forma paralela só pode ser executado pelo processador para qual foi escrito. Deste ponto em diante, computador sequencial se refere a um código sequencial executado em um processsador comum e computador paralelo se refere a código paralelo sendo executado em uma máquina paralela.

Um exemplo muito utilizado é a previsão do tempo. A solução para este tipo de problema demanda uma quantidade muito grande de cálculos aritméticos em uma quantidade ainda maior de dados colhidos por sensores espalhados em terra, água ou ar.As variáveis neste problema são "subdivisões" da atmosfera, e o número destas "subdivisões" é muito grande. Se um computador sequencial fosse utilizado, o cálculo para prever a condição climática do dia seguinte levaria mais do que um dia, o que é totalmente inviável. Porém, se um computador paralela for utilizado, o problema pode ser resolvido em questão de minutos.

O conceito de programação paralela é que, se para resolver um problema, um computador com apenas um processador leva um tempo $t$, utilizando $n$ processadores este tempo se reduziria a $t/n$. Este resultado é apenas teórico, ou pertencente a uma pequena classe de problemas específicos, os chamados "problemas embaraçosamente paralelos", ou *embarassingly parallel pro-*

*blems*, que são problemas que podem ser paralelizáveis sem muito esforço, por exemplo o problema de transformações geométricas em uma imagem, onde os dados de cada processador são totalmente independentes, evitando comunicação ou ociosidade excessiva.

#### 2.3.0.1 *Modelos de Programação Paralela*

Existem três principais modelos para a programação paralela, que se diferem principalmente no tipo de conexão entre os processadores e as memórias e sua distribuição. Estes modelos, conforme [60], são:

- Multiprocessador de Memória Compartilhada: neste modelo, muito parecido com um computador convencional, que é organizado com a conexão direta de um processador a uma memória principal, todos os processadores são conectados a uma rede de interconexão que também é ligada a todos os módulos da memória principal. Assim todos os processadores tem acesso a uma memória compartilhada. Este modelo tem como facilidade o compartilhamento de dados, já que todos os processadores têm acesso aos mesmos dados, porém o hardware é complexo demais para controlar todas as leituras e escritas concorrentes, e não consegue atingir uma boa velocidade de acesso à memória. Para minimizar este problemas, hierarquias de memória são implementadas, fazendo com que processadores possuam memórias mais próximas, agilizando o processo de acesso. Memórias *cache* também são utilizadas;

- Multicomputador com Troca de Mensagens: vários computadores (processador e memória) são conectados através de uma rede (existem várias topologias de rede, que serão citadas posteriormente), ou seja, um processador tem acesso a apenas sua memória, e não a dos outros. Para que um dado de outro processador seja lido, mensagens devem ser criadas e enviadas pela rede. A escalabilidade é um ponto forte deste modelo em comparação ao anterior. Para que seu tamanho seja aumentado basta conectar um computador a mais na rede, já no modelo anterior uma modificação no hardware deve ser feita.

- Memória Compartilhada Distribuída: visando unir os pontos positivos dos dois modelos anteriores, este foi criado. Tem a mesma forma de um multicomputador com troca de mensagens, porém o espaço de endereçamento de memória é um só, ou seja, um processador pode acessar a memória de outro, pois, virtualmente estes possuem uma memória compartilhada. Porém este modelo apenas esconde a troca de mensagens, pois ao acessar um dado de uma outra memória que não a sua própria, mensagens devem ser enviadas pela rede.

O multicomputador com troca de mensagens pode ser conectado de diversas maneiras, desde uma simples lista de computadores a um complexo hipercubo de várias dimensões. Cada problema possui uma topologia ideal,

que se aproveita de características desta rede para realizar comunicações. Uma topologia muito utilizada atualmente é a de totalmente conectados, ou *"fully connected"*, onde todos os computadores estão conectados entre si. Esta topologia não era usada no passado pois a velocidade e latência das redes atuais é muito melhor do que as antigamente utilizadas, porém as topologias de grade, *torus* e outras ainda são utilizadas para aplicações específicas.

No modelo de multiprocessador de memória compartilhada, existem as *GPUs*, que foram criadas para processar imagens em computadores, removendo uma grande quantia de trabalho do processador principal. Quando uma imagem deve ser processada, diversos *pixels* devem ser processados e, para cada um deles, um mesmo código deve ser executado. Por isto as *GPUs* foram criadas já com modelos de programação paralela.

Tentando utilizar as *GPUs* para processar outros dados que não imagens, criou-se o conceito de *GPGPU* [61], que separa a execução do código em diversas *threads* concorrentes. Uma *thread* é uma unidade de execução, menor que um processo, que possui um estado próprio, ou seja, possui memória e registradores, como ponteiros de instrução e pilha, e estes dados são locais, não são compartilhados. Normalmente em códigos executados em *GPGPUs* existem centenas ou milhares de *threads* sendo executadas concorrentemente, cooperando através de uma memória acessível a todas ou a apenas um grupo restrito. O conceito de *GPGPUs* é melhor detalhado na seção seguinte.

### 2.3.1 *GPGPUs*

As *GPUs* [62] são dispositivos criados para gerar imagens e vídeos, que requerem o processamento de um grande volume de dados, executam um número muito grande de instruções em vários núcleos, ou *cores*, ultrapassando a vazão de um processador principal em várias vezes, mesmo normalmente possuindo uma taxa de *clock* muito menor e seus *cores* sendo muito mais simples. Instruções matemáticas são predominantes em execuções em *GPUs*, já que para calcular, por exemplo, a interseção de um raio com objetos em um traçador de raios, a maioria das operações são de lógica e aritmética e, ainda, os dados que são processados por diferentes núcleos são independentes.

Observando este comportamento em uma *GPU*, o conceito de *GPGPU* foi criado, para que se pudesse aproveitar estes cálculos paralelos em vários núcleos e numa vazão maior que a de uma *CPU*, para resolver problemas gerais, e não somente problemas de geração de imagens.

Para que a execução de um problema tenha um bom desempenho em uma *GPU*, deve-se considerar as propriedades a seguir:

- Volume de dados grande, já que as *GPUs* trabalham com bilhões de *pixels* por segundo e cada *pixel* é submetido a centenas de operações aritméticas. Problemas cujo cálculo da solução é lento devido à alta complexidade de execução de seu algoritmo, e não à quantidade de dados, não são bons candidatos a serem executados em uma *GPGPU*

já que possivelmente não serão capazes de utilizar a alta capacidade de paralelização deste hardware.

- Independência de dados, pois na geração de uma cena, por exemplo, os vértices podem ser calculados independentemente, evitando dependências de execução, ou seja, esperar que um certo cálculo seja realizado para outro poder executar.

- Escalabilidade: se mais núcleos forem adicionados ao hardware, melhor desempenho o programa tem.

- Vazão: em uma *GPU*, os processadores utilizam *pipelines*. Como consequência, a resolução de um conjunto de cálculos pode levar um tempo grande. Porém vários conjuntos de cálculos são realizados em um determinado período de tempo, devido ao processamento paralelo. Problemas que serão executados em uma *GPGPU* devem priorizar uma alta vazão de resultados, e não uma baixa latência para obter um resultado.

Para que um problema seja executado em uma *GPGPU*, o programador deve separar pequenos trechos de código que podem ser executados paralelamente e escrevê-los em uma linguagem própria. Neste trabalho, a linguagem utilizada foi a linguagem CUDA C [63], desenvolvida pela empresa NVIDIA. A linguagem CUDA C é é similar à linguagem C, mas com algumas adições como a separação de trechos de códigos que serão executados na *CPU* ou *GPU*.

As *GPUs* da NVIDIA com capacidade de executar código CUDA estão divididas em várias versões, conforme a arquitetura das *GPUs* foi evoluindo. Estas versões são chamadas de *Compute Capability* ou capacidade de computação. As primeiras *GPUs* com CUDA tinham *Compute Capability* ou (CC) 1.0 e eram bem limitadas. Hoje há placas com CC 3.0, com características muito mais sofisticadas do que as anteriores e, em breve, serão lançadas placas com CC 3.5.

Em CUDA C, um *kernel* é um código que será executado paralelamente na *GPU*. Para se executar um *kernel*, um *grid* deve ser configurado. Este *grid* nada mais é do que uma configuração de como as *threads* serão organizadas na execução, através de uma matriz de até três dimensões de blocos (para placas com CC 1.x são no máximo duas dimensões), e cada um destes blocos é uma matriz de até três dimensões de *threads*. Um bloco de *threads* é um conjunto de *threads* que podem cooperar entre si através de ferramentas de sincronização e compartilhamento de memória, *threads* de diferentes blocos não se comunicam diretamente nem podem sincronizar, apenas pela memória global. O lançamento de um *kernel* é naturalmente assíncrono, ou seja, assim que o código sequencial lança o *kernel*, a execução sequencial continua. Se for preciso utilizar o valor calculado pelo *kernel*, uma instrução de barreira deve ser utilizada para aguardar o fim da execução do *kernel*. A Figura 2.5 representa a execução de um código com lançamento de *kernel*.

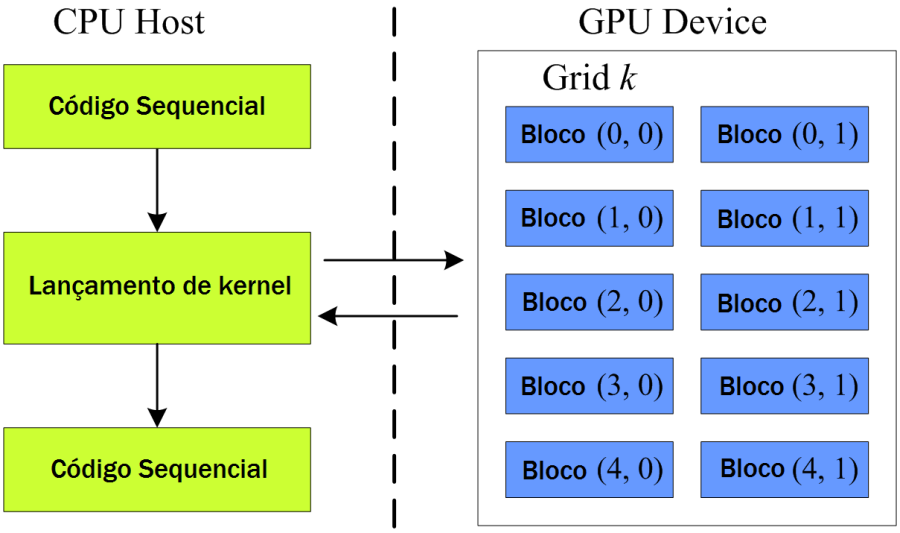



Figura 2.5: Representação de um programa sendo executado em CUDA e em CPU.

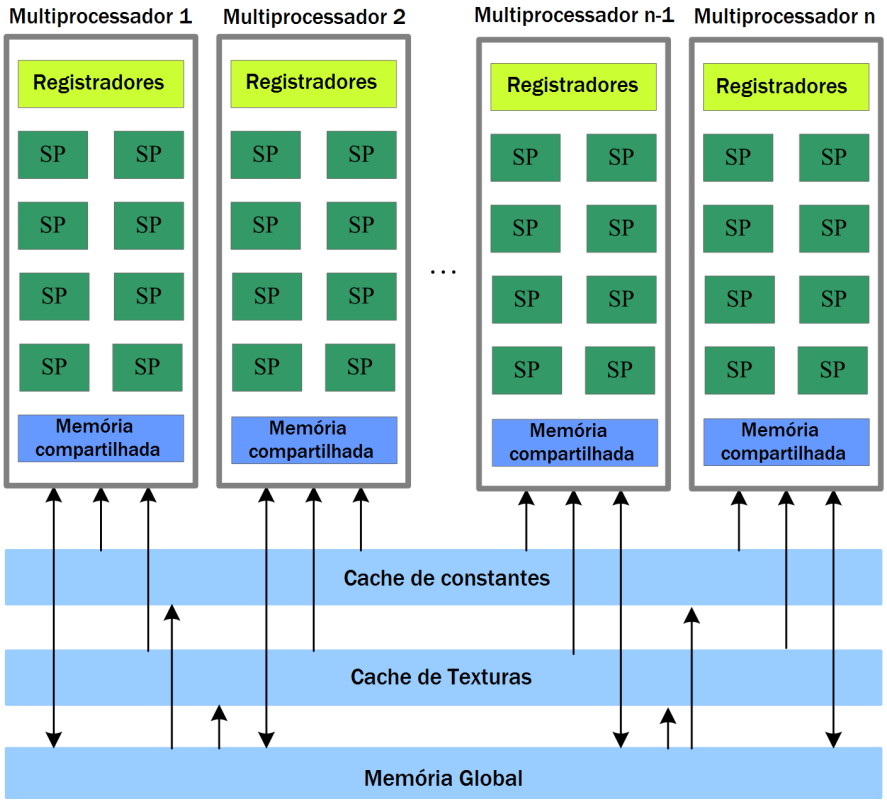



Figura 2.6: Organização dos multiprocessadores de uma GPU da NVIDIA com CUDA.

Quando um *kernel* é lançado, os blocos que foram definidos são divididos entre os multiprocessadores (*streaming multiprocessors* ou SM), representados na Figura 2.6, que é a unidade de computação de um bloco e contém 8

CUDA *cores* (32 na arquitetura Fermi), unidades *load/store*, escalonadores de *warp* entre outros. Um *warp* é um conjunto de 32 *threads* e será melhor definido a seguir. Cada bloco reside em apenas um SM, e durante a execução de um *kernel* ele não migra para outros SMs. Um SM pode conter mais de um bloco, e daqui vem a escalabilidade do CUDA, se uma GPU possui apenas um SM com 32 blocos, se o número de SMs for aumentado para 4, existirão 4 SMs com 8 blocos cada, aumentando o paralelismo da execução.

As *threads* são organizadas em *warps*, que são conjuntos de 32 *threads*, e estas *threads* não mudam de *warp* durante a execução. Os *warps* são divididos de maneira lógica: *threads* de 0 até 31 pertencem ao primeiro *warp*, *threads* de 32 até 63 pertencem ao segundo *warp* e assim por diante. A execução de um bloco é sempre feita em *warps*, se o número de *threads* não for múltiplo de 32, *threads* vazias, que não alteram o resultado do programa, são inseridas no *warp*. Por isso devem-se evitar instruções de desvio que, em um mesmo *warp*, podem tomar caminhos diferentes. Por exemplo, suponha que um *warp* está executando uma instrução condicional. Se apenas na primeira *thread* a condição for verdadeira, o *warp* inteiro deverá executar o código de condição verdadeira, porém todas menos a primeira *thread* são desativadas para garantir a corretude do algoritmo. Após a execução de condição verdadeira, o *warp* inteiro executará o código de condição falsa com a primeira *thread* desativada.

As instruções são despachadas por *warp* pelo escalonador do SM, ou seja, a cada instrução despachada um *warp* é alocado e executado. Se um dos operandos de uma instrução despachada não está pronto, o *warp* é parado e, se possível, acontece uma troca de contexto, fazendo com que o *warp* parado volte para a fila do escalonador e outro *warp* com operandos prontos possa ser executado. Como um SM pode conter mais de um bloco, se o *warp* que será executado é diferente do que estava sendo executado anteriormente, uma troca de contexto de bloco deve ser feita pelo SM. O contexto de um bloco é o conjunto de registradores e memória compartilhada e diz-se que um bloco está ativo se algum de seus *warps* está sendo executado pelo seu SM. Enquanto um bloco estiver ativo, ele continua ativo até que todas suas *threads* sejam executadas completamente.

# 3

# AMBIENTE DE DESENVOLVIMENTO E TESTE

Neste trabalho, foram tratados os problemas da mochila multidimensional, mochila binária e problema quadrático de alocação. Nos capítulos a seguir estes problemas são definidos. Utilizando otimização de colônia de formigas, também são descritos algoritmos para a solução desses problemas.

Os algoritmos foram implementados utilizando C++ e CUDA. A máquina usada pra executar os programas contém um i5 — 2500K com *clock* padrão em 3.3GHz, 8GB de memória principal e uma GPU NVIDIA GTX570. O sistema operacional é Microsoft Windows 7 64 *bits*. A GTX570 possui 480 cores CUDA, com *clock* de processador 1.464GHz e 1280MB de memória. Mais informações sobre esta GPU podem ser obtidas no *website* da NVIDIA [64]. Todos os algoritmos foram codificados no Visual Studio 2010, com o Toolkit CUDA e o *plugin* Nsight da NVIDIA.

# 4

# PROBLEMA DA MOCHILA

O problema da mochila (*knapsack problem*, ou KP) [9] é um dos problemas de programação linear mais simples de ser expressado. Problemas de programação linear são problemas de otimização nos quais a função objetivo e as restrições são todas lineares. Este problema pode ser enunciado em linguagem natural da seguinte forma:

*Um viajante levará consigo apenas uma mochila para sua viagem. Sua mochila possui uma dada capacidade e deve ser preenchida com diferentes tipos de objetos que lhe serão úteis durante a viagem. Cada objeto ocupa uma certa capacidade da mochila e tem um dado valor para o viajante. Quais objetos devem ser levados pelo viajante em sua mochila de forma a maximizar o valor do seu conteúdo?*

Existem diversas variações do problema da mochila, algumas são descritas a seguir:

PROBLEMA DA MOCHILA BINÁRIA UNIDIMENSIONAL

Neste problema, a mochila possui apenas uma dimensão, assim pode-se imaginar como se cada objeto possui apenas um peso e um valor, ambos valores pertencentes ao conjunto dos números naturais. Para cada um dos $n$ objetos $j$, $x_j = 1$ se este objeto encontra-se na mochila, caso contrário $x_j = 0$. $p_j$ é o seu valor, $w_j$ é o seu peso e $W$ é a capacidade da mochila. A solução é encontrar um vetor $x$ $(x_1, x_2, ..., x_n)$ que satisfaça:

$$\begin{aligned} &\max \sum_{j=1}^{n} p_j x_j (p_j \in \mathbb{N}^*) \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_j \leqslant C (w_j, C \in \mathbb{N}^*) \\ &\qquad\qquad x_j \in \{0, 1\}, 1 \leqslant j \leqslant n \end{aligned} \tag{4.1}$$

PROBLEMA DA MOCHILA MULTIDIMENSIONAL

Neste problema, a mochila possui mais de uma dimensão, ou seja, ao escolher um objeto não só o peso é levado em consideração. Pode-se, por exemplo, interpretar como peso e volume se o problema possuísse 2 dimensões. Para cada objeto $j$, $x_j = 1$ se este objeto encontra-se na

mochila, $p_j$ é o seu valor, $w_{ji}$ é o custo do objeto $j$ na dimensão $i$ e $b_i$ é a capacidade da mochila na dimensão $i$. A solução é encontrar um vetor $x$ $(x_1, x_2, ..., x_n)$ que satisfaça:

$$\begin{aligned} &\max \sum_{j=1}^{n} p_j x_j, \\ &\text{sujeito a} \sum_{j=1}^{n} w_{ij} x_j \leqslant b_i, \forall i = 1, ..., m, \\ &\qquad\qquad x_j \in \{0, 1\}, 1 \leqslant j \leqslant n \end{aligned} \tag{4.2}$$

PROBLEMA DA MOCHILA LIMITADA

Dado o conjunto de $n$ objetos, para cada objeto $j$ é dado seu valor $p_j$, seu peso $w_j$ e o número de objetos iguais disponíveis (cópias) $b_j$. Também deve ser dada a capacidade $C$ da mochila. No problema da mochila 0-1, o valor $x_j$ era sempre 0 ou 1, que significava que o objeto esta dentro da mochila ou não, porém, neste problema, $x_j$ pode estar entre 0 e $b_j$ ($0 \leqslant x_j \leqslant b_j$), significando que o objeto pode não estar na mochila ou até $b_j$ cópias deste objeto podem estar na mochila. A solução é encontrar um vetor $x$ $(x_1, x_2, ..., x_n)$ que satisfaça:

$$\begin{aligned} &\max \sum_{j=1}^{n} p_j x_j, \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_j \leqslant C. \\ &\qquad x_j \leqslant b_j, 1 \leqslant j \leqslant n \end{aligned} \tag{4.3}$$

PROBLEMA DA MOCHILA ILIMITADA

Igual ao problema da mochila limitada, porém assume-se que cada objeto possui infinitas cópias, ou seja, para cada objeto $j$, $b_j = \infty$. A representação em programação linear é a mesma, a única mudança é o limite do valor de $b_j$:

$$\begin{aligned} &\max \sum_{j=1}^{n} p_j x_j, \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_j \leqslant C. \\ &\qquad x_j \in \mathbb{Z}, 1 \leqslant j \leqslant n \end{aligned} \tag{4.4}$$

PROBLEMA DA SOMA DE SUBCONJUNTOS

Igual ao problema da mochila 0-1, porém aqui o peso de um objeto é sempre igual ao seu valor, ou seja, para um objeto $j$, $p_j = w_j$. Representado em programação linear:

$$\begin{aligned} &\max \sum_{j=1}^{n} w_j x_j, \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_j \leqslant C \\ &x_j \leqslant 0, 1, 1 \leqslant j \leqslant n \end{aligned} \tag{4.5}$$

*change making knapsack problem*

Dado o conjunto de $n$ objetos e, para cada objeto $j$, seu peso $w_j$, seu valor $p_j$, o número de cópias do objeto $j$ $b_j$ e a capacidade da mochila $C$, determinar um vetor $(x_1, x_2, ..., x_n), x_j \in \mathbb{N}$ que satisfaça:

$$\begin{aligned} &\max \sum_{j=1}^{n} p_j x_j, \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_j = C \\ &x_j \leqslant b_j, 1 \leqslant j \leqslant n \end{aligned} \tag{4.6}$$

Este problema é facilmente entendido quando visto como o problema de trocar uma quantia de dinheiro em células pela quantidade máxima de moedas possível, mantendo o mesmo valor monetário, no caso em que $p_j = 1, \forall j$.

PROBLEMA DA MOCHILA BINÁRIA MÚLTIPLA

Igual ao problema da mochila binária unidimensional, porém existem múltiplas ($m$) mochilas, cada mochila $i$ com sua respectiva capacidade positiva e inteira $C_i$. Para cada objeto $j$, $x_{ij} = 1$ se este objeto encontra-se na mochila $i$, e $x_{ij} = 0$ caso contrário. É igual ao problema da mochila multidimensional. A solução é encontrar a matriz $x_{ij}$ que satisfaça:

$$\begin{aligned} &\max \sum_{j=1}^{n} \sum_{i=1}^{m} p_j x_{ij}, \\ &\text{sujeito a} \sum_{j=1}^{n} w_j x_{ij} \leqslant C_i, 1 \leqslant i \leqslant m \\ &\sum_{i=1}^{m} x_{ij} \leqslant 1, 1 \leqslant j \leqslant n \\ &x_{ij} \in \{0, 1\}, 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n \end{aligned} \tag{4.7}$$

Em [10] é demonstrado que todos os problemas apresentados acima fazem parte da classe dos problemas NP-difíceis [65].

## 4.1 MOCHILA 0-1 MULTIDIMENSIONAL (*mkp*)

Este trabalho trata do problema da mochila 0-1 multidimensional, também chamada de apenas mochila multidimensional ou MKP, que é um caso geral de programação linear binária, e é uma generalização da mochila binária, onde o número de dimensões é 1 ($m = 1$). Este problema pode ser visto como uma mochila com várias dimensões ($m \geqslant 2$) ou várias mochilas de uma única dimensão. Historicamente, os primeiros exemplos foram exibidos por Lorie e Savage [66] e por Manne e Markowitz [67] como problemas de análise de orçamentos de capital, que é um processo que decide se um investimento a longo prazo vale a pena. O MKP é um modelo de alocação de recursos que pode ser descrito como

$$\begin{aligned} \max z &= \sum_{j=1}^{n} c_j x_j, \\ \text{sujeito a } &\sum_{j=1}^{n} a_{ij} x_j \leqslant b_i, \forall i = 1, ..., m, \\ & x_j \in \{0, 1\}, 1 \leqslant j \leqslant n \end{aligned} \tag{4.8}$$

onde $n$ é o número de itens e $m$ é o número de restrições da mochila (ou o número de mochilas binárias), cada uma com capacidade $b_i$ ($1 \leqslant i \leqslant m$). Cada item $j$ requer $a_{ij}$ recursos na dimensão (ou mochila) $i$ e tem um valor $c_j$ se for incluído. O objetivo é encontrar um subconjunto de itens que tenha a soma dos valores máxima, sem exceder a capacidade de cada dimensão. Por natureza, pode ser assumido sem perda de generalidade, que $c_j > 0$ para cada objeto, ou seja, nenhum objeto terá valor negativo, $b_i > 0$ para cada dimensão, pois se uma dimensão não tem capacidade, pode-se removê-la junto com todos os objetos que necessitam de pelo menos um recurso nesta dimensão, $0 \leqslant a_{ij} \leqslant b_i$, pois nenhum objeto escolhido pode ocupar recursos negativos, que é a mesma coisa que aumentar a capacidade da mochila na dada dimensão, nem ocupar mais recursos do que os disponíveis, ou então este objeto poderia ser removido do problema. Ainda, $\sum_{j=1}^{n} a_{ij} > b_i$ para todo $i \in N$ e $j \in M$, ou algumas ou todas as variáveis poderiam ser fixadas em 0 ou 1). Qualquer instância do MKP com qualquer $a_{ij} = 0$ pode ser substituída por uma instância equivalente com somente entradas positivas que tem soluções ótimas idênticas.

O grande domínio de aplicações do MKP contribuiu muito para sua fama [68]. No trabalho de referência de Lorie e Savage [66], o MKP foi intensamente investigado para orçamentos de capital e aplicações para seleção de projetos [69, 70, 71]. O MKP também foi introduzido para modelar problemas como corte de estoque [72], problemas de carregamento [73, 74], política de investimentos no setor de turismo de um país em desenvolvimento [75], atribuição de bancos de dados e processadores em processamento de dados distribuídos [76], distribuição de compras em um veículo com vários compartimentos [77] e votação de aprovação[78]. Mais recentemente, o MKP foi usado para modelar a administração diária de um satélite de sensoriamento a distância, que é a análise do planeta feita por aparelhos,

como o SPOT, que consistia em decidir, a cada dia, que fotografias seriam tentadas no dia seguinte [79].

Um dos primeiros algoritmos exatos para o MKP foi apresentado por Gilmore e Gomory [72] e utilizava programação dinâmica. Mais tarde, algoritmos híbridos foram criados, misturando programação dinâmica e *branch-and-bound* com heurísticas com baixo consumo de tempo e limitantes obtidos através de programação linear (*LP bounds*). Devido a quantidade excessiva de memória que estes algoritmos necessitam, apenas instâncias com baixo valor de $n$ podem ser resolvidas. Existem outros métodos exatos, como enumeração, programação linear, relaxação Lagrangeana, otimização de subgradiente e outros, mas nenhum soluciona todas as instâncias do MKP eficientemente. Também existem frameworks comerciais que permitem uma implementação utilizando *branch-and-cut* para a solução de problemas em geral, entre eles o MKP. Entre estes frameworks, os mais utilizados são o CPLEX [80], OSL [81] e XPRESS-MP [82]. Estes softwares podem em algumas instâncias não encontrar a solução ótima, se isso acontecer, uma solução perto da ótima é retornada.

Na subseção a seguir é apresentado o algoritmo de otimização de colônia de formigas utilizado para resolver o MKP, com detalhes de implementação. Na subseção seguinte os resultados obtidos são apresentados.

### 4.1.1 *Colônia de formigas para o Problema da Mochila Multidimensional*

O algoritmo ACO implementado para resolver o problema da mochila multidimensional é mostrado abaixo e foi baseado no algoritmo apresentado em [83], com algumas modificações devido a sua implementação ser em CUDA. O Algoritmo 1 apresenta o pseudo-código do algoritmo implementado.

Ao invés de representar cada formiga como uma *thread*, o que é a modelagem mais simples e intuitiva, a implementação representa uma formiga como um bloco quando o código das formigas está sendo executado. Quando passos globais como a atualização e inicialização estão sendo executados, cada bloco deixa de representar uma formiga e passa a representar uma colônia.

Na maioria dos sistemas de colônias de formigas, existe apenas uma colônia: um conjunto de formigas, uma tabela de feromônio e possivelmente variáveis que controlam o comportamento das formigas. Isto acontece porque a maioria dos algoritmos é implementado sequencialmente, se duas colônias fossem implementas, o tempo de execução iria dobrar. Mas como GPGPUs estão sendo utilizadas e existem muitos recursos paralelos para usar, ao invés de criar muitas formigas que utilizam apenas uma tabela de feromônio, elas podem ser divididas em duas ou mais colônias. Outro ponto interessante para o uso de várias colônias é que a atualização do feromônio é elitista: somente a melhor solução gerada pelas formigas em uma dada rodada é utilizada. Se muitas formigas utilizam a mesma tabela, muito do trabalho realizado por elas não é usado, então pode ser melhor dividi-las em mais

colônias. A tabela de feromônio possui $n$ posições, uma para cada objeto, e utiliza o conceito de $min - max$, o que significa que existe um valor mínimo, que é chamado de $t_{min}$, e um valor máximo, que é chamado de $t_{max}$, para cada posição da tabela. No começo do algoritmo, todas as posições são inicializadas em $t_{max}$, assim cada objeto tem um mesmo fator de feromônio.

A desejabilidade de um objeto é um valor que representa o quão bom um objeto é para uma certa mochila. Quando uma formiga quer adicionar um objeto à mochila, ela calcula a desejabilidade de cada objeto, e então escolhe um deles probabilisticamente. Assim, quanto maior a desejabilidade de um objeto, maior a probabilidade de ele ser inserido na mochila. Define-se $S_k$ o estado do algoritmo, este estado contém todos os dados utilizados na execução, como o custo e o valor de cada objeto, a tabela de feromônio e outros. A desejabilidade de um objeto é calculado da seguinte maneira:

$$D_i = \tau_{factor}(i, S_k)^{\alpha} + \eta_{factor}(i, S_k)^{\beta} \tag{4.9}$$

Onde $\tau_{factor}(i, S_k)$ é o fator feromonal de um objeto (valor da sua posição na tabela de feromônios) e $\eta_{factor}(i, S_k)$ é o fator heurístico do objeto, que é algo parecido como custo-benefício, e é calculado como mostrado abaixo. Os valores de $\alpha$ e $\beta$ são parâmetros que indicam o peso de cada fator. Se deseja-se que o fator feromonal seja mais determinante que o fator heurístico, deve-se colocar $\alpha > \beta$.

$$\eta_{factor}(i, S_k) = p_i / \sum_{j=1}^{m} (w_{ij}/cr_j) \tag{4.10}$$

onde $p_i$ é o valor do objeto $i$, $w_{ij}$ é o custo do objeto $i$ na dimensão $j$ e $cr_j$ é a capacidade restante na mochila na dimensão $j$. Ao calcular o fator heurístico de um objeto, cada formiga verifica se este objeto cabe na mochila (um objeto não cabe se $w_{ij} > cr_j$ para qualquer $j$) ou se ele já está na mochila. Se o objeto não cabe ou já está na mochila, sua desejabilidade se torna 0, já que não é possível adicioná-lo.

Quando uma formiga deseja adicionar um objeto à mochila, ela deve calcular a desejabilidade de $n$ objetos e escolher um deles. Para isso, cada formiga, representada por um bloco, lança $t$ *threads*, onde $t$ não é maior que $n$. Poderiam ser lançadas $n$ *threads*, uma para cada objeto, porém cada bloco possui um limite de *threads* que podem ser lançadas. Este limite é 512 para GPUs com *compute capability* até 2.0, e 1024 para GPUS 2.0 e acima), e este limite é facilmente alcançado por problemas de tamanho grande.

Cada uma destas $t$ *threads* calcula a desejabilidade de aproximadamente $n/t$ objetos e, em seguida, multiplica o valor obtido por um número aleatório entre 0 e 1. Após o cálculo, uma redução binária em paralelo é feita no vetor para encontrar o maior valor calculado. A multiplicação por um número aleatório é feita para que se possa utilizar o método *I-Roulette* de [15]. Este método substitui a tradicional roleta para escolher um valor por um método paralelo. No método tradicional, cada objeto tem uma probabilidade de ser escolhido, que seria $D_i / \sum_{j=1}^{n} D_j$, então um valor uniforme

seria gerado, e seu valor decrementado pela probabilidade de cada objeto até que ele fosse igual ou menor a zero. Quando isto acontecesse, o último objeto cuja probabilidade foi subtraída seria o objeto escolhido. É fácil ver que a complexidade deste método é $O(n)$, visto que existe a possibilidade de o vetor inteiro ter que ser subtraído e o último elemento ser escolhido. O método *I-Roulette* é mais sofisticado. A probabilidade de cada objeto é multiplicada por um número uniforme aleatório, multiplicado por um valor binário, que neste caso não foi utilizado, já que a implementação já verifica se o objeto não cabe ou já está na mochila, e guardado em um vetor. Após a redução binária tem-se o maior valor do vetor, a posição deste valor é o objeto escolhido. Como todo o algoritmo pode ser executado em um passo se houverem $n$ *threads* e a redução consegue encontrar o valor máximo em $O(\log n)$, a complexidade de escolher um objeto dentre os $n$ é $O(\log n)$.

Note que cada *thread* pode calcular a desejabilidade de mais de um objeto e o número de objetos pode ser muito maior que o número de *threads*. Porém um vetor de tamanho $t$ é utilizado, então não se consegue armazenar todos os valores dos objetos. Ao invés disto, antes que uma *thread* calcule o valor do primeiro objeto, um valor muito pequeno é escrito em sua posição, então, cada vez que o valor de um objeto é calculado, gulosamente armazena-se este valor apenas se ele é maior que o que está armazenado. Então se uma *thread* irá calcular $n/t$ desejabilidades, apenas a maior delas pode ser escolhidas, as outras são descartadas.

O passo de atualização da tabela de feromônio é simples: primeiro atualizam-se os valores de todas as posições multiplicando-os por um certo número, $1-$ o fator de evaporação. Se o fator de evaporação for igual a $1$, todo o feromônio evapora e o algoritmo se torna um gerador de soluções aleatórias, se for $0$ nenhum feromônio evapora. Após a evaporação, cada colônia atualiza a tabela utilizando apenas a melhor solução gerada na rodada atual por suas formigas. E apenas o valor do feromônio relativo às posições dos objetos que estão nesta solução são incrementados. O valor incrementado é calculado da seguinte maneira: $1.0/(1.0 + \mathtt{Best} - \mathtt{Roundbest})$, onde $\mathtt{Best}$ é o valor da melhor solução gerada pela colônia desde o começo do algoritmo, e $\mathtt{Roundbest}$ é o valor da melhor solução gerada na rodada atual.

A complexidade deste algoritmo é $O(n^2 * m)$ para cada formiga em cada rodada, pois a cada passo de construção de mochila, até $n$ objetos podem ser inseridos e, para inserir um objeto, a desejabilidade de todos os $n$ objetos devem ser calculados, e cada desejabilidade utiliza o custo daquele objeto em todas as $m$ dimensões.

### 4.1.2 *Resultados experimentais*

Como visto, o algoritmo de colônia de formigas para a mochila multidimensional necessita de um conjunto de parâmetros de entrada, que modificam alguns comportamentos da execução. Várias configurações foram testadas, e apenas a melhor delas é mostrada nos resultados. O par de número de formigas / número de colônias testadas foram: 8/1, 32/2, 128/4 e

**Algoritmo 1** Algoritmo ACO para o problema da mochila multidimensional.

1: Inicialize cada posição da tabela de feromônio com $t_max$
2: **para** i de 1 até o número máximo de rodadas **faça**
3: **para** cada formiga $k = 1, \ldots, m$ **faça**
4: Adicione um objeto aleatório à mochila.
5: Adicione objetos à mochila até que nenhum outro objeto caiba, utilizando a fórmula de desejabilidade e o I-Roulette para escolher os objetos.
6: **fim-para**
7: Ajuste a melhor solução de cada colônia.
8: Atualiza a tabela utilizando a melhor solução da colônia e a melhor solução gerada nesta rodada pela colônia.
9: **fim-para**

256/8. Em todos as execuções, o tempo necessário para executar as configurações com oito formigas em uma colônia e trinta e duas formigas em duas colônias foram iguais, porém com oito formigas as soluções geradas eram piores. Isto acontece porque a GPU executa vários multiprocessadores ao mesmo tempo, e este número de multiprocessadores é maior que 32, então executar qualquer número de formigas até 32 necessita de uma mesma quantidade de tempo.

Os pesos $\alpha$ e $\beta$ foram testados em 1 e 1, 4 e 1, 1 e 4. Porque estes parâmetros representam o peso dos fatores, 4 e 4 não foi usado pois tem o mesmo peso de 1 e 1. Após os testes, verificou-se que quando $\alpha$ and $\beta$ eram 4 e 1, respectivamente, os melhores resultados eram obtidos. Os parâmetros da tabela de feromônio $t_{min}$ e $t_{max}$ foram variados, porém não alteraram significativamente os tempos de execução médios, então foram fixados em 0.01 e 6, respectivamente. O último parâmetro é o fator de evaporação. Foram testados valores de 0.1, 0.3 e 0.7, e os melhores resultados foram obtidos quando o valor era 0.1.

As instâncias do MKP utilizadas foram as instâncias da biblioteca OR [84], que é uma coleção de problemas de programação inteira que contém, entre outros, instâncias do MKP. Existem dois conjuntos de problemas do MKP na biblioteca OR. O primeiro contém 55 problemas pequenos e não-correlatos, coletados de trabalhos antigos de Fréville and Plateau [85], com problemas variando o número de objetos de 6 até 105 e o número de dimensões variando de 2 até 30. Este conjunto é fácil de resolver, o CPLEX v6.5.2 [86] foi capaz de resolver todos os problemas em menos de três segundos de tempo de CPU. A maioria das soluções ótimas para estes problemas está contida em trabalhos anteriores de Shih [74] e Gavish e Pirkul[87].

O segundo conjunto contém instâncias mais difíceis [88], e foram geradas utilizando o procedimento sugerido por Fréville e Plateau [89]. O número de dimensões ($m$) foi fixado em 5, 10 e 30, e o número de objetos($n$) fixado em 100, 250, e 500. Para cada possível par de $n$, $m$, trinta instâncias foram geradas, dez para cada fator de "aperto". O fator de "aperto" de uma instância de-

termina a capacidade da mochila em cada dimensão em função da soma do peso de todos os objetos para aquela dada dimensão, assim, quanto menor o fator de "aperto", menos objetos a mochila pode suportar. Mais informações sobre a biblioteca OR e como ela foi gerada pode ser encontrada em [68]

Como a maioria dos trabalhos que utilizam a biblioteca OR possuem referências à endereços da internet que originalmente continham as instâncias, porém não existem mais, as instâncias foram retiradas de outro endereço [90], que contém todas as instâncias e suas soluções ótimas se conhecidas ou melhores soluções encontradas até o momento. Também contém referências de quais trabalhos encontraram os melhores valores para cada instância.

Quando uma instância não possui valor ótimo provado, a melhor solução descoberta foi considerada a ótima. A qualidade de uma solução significa o quão próximo ela está da ótima.

O tempo mostrado nas tabelas é medido em milissegundos e foi obtido através da média aritmética após 10 execuções para cada instância, assim como todos os outros dados. Para medir a distância de uma solução à ótima, a medida de porcentagem de desvio médio (PDM) é utilizada. Este valor pode ser calculado da seguinte maneira: 100∗ (valor da solução obtida − valor da solução ótima) / valor da solução ótima. Quanto menor o PDM, melhor. Se o PDM é igual a zero, significa que nas 10 execuções a solução ótima foi encontrada. Também foram contabilizados em quantas das 10 execuções a solução ótima foi encontrada, este valor é representado pelas colunas OPT. Da Tabela 4.1 até 4.6 são mostrados os resultados obtidos. Na Figura 4.1 pode-se ver um gráfico da qualidade da solução pelo número de rodadas executadas em uma das instâncias executadas.

Tabela 4.1: Resultados das execuções para as instâncias do primeiro conjunto com 32 formigas em 2 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| Nome do conjunto de problemas | Número de problemas | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| HP | 2 | 2 | 2.8128 | 0 | 25 | 1.1306 | 1 |
| PB | 6 | 2 | 2.5769 | 10 | 26 | 0.8756 | 21 |
| PETERSEN | 6 | 2 | 0.4682 | 31 | 24 | 0.2877 | 42 |
| SENTO | 2 | 7 | 0.5443 | 0 | 75 | 0.1878 | 1 |
| WEING | 8 | 3 | 0.0520 | 49 | 34 | 0.0009 | 77 |
| WEISH | 29 | 4 | 0.7856 | 28 | 43 | 0.4119 | 93 |

Para comparar os resultados deste trabalho com os resultados de trabalhos da literatura ([7] de 2006, [91] de 2010, [92] de 2010 e [93] de 2012) a configuração de 256 formigas e 100 rodadas foi escolhida por ter qualidade e principalmente tempo consumido mais compatíveis, esta configuração será chamada de ACO a partir deste ponto com a finalidade de facilitar a leitura.

As Tabelas 4.7 e 4.8 mostram os dados da configuração de 256 formigas deste trabalho, que obteve tempos e qualidades de solução mais compatíveis e será chamado de ACO a partir deste ponto, e os resultados encontrados na literatura recente. Apenas o segundo conjunto será comparado pois é o

Tabela 4.2: Resultados das execuções para as instâncias do primeiro conjunto com 128 formigas em 4 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| Nome do conjunto de problemas | Número de problemas | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| HP | 2 | 4 | 2.3228 | 0 | 42 | 1.0758 | 5 |
| PB | 6 | 4 | 2.8139 | 10 | 44 | 1.3633 | 19 |
| PETERSEN | 6 | 4 | 0.5535 | 30 | 40 | 0.3155 | 33 |
| SENTO | 2 | 14 | 0.6140 | 0 | 146 | 0.3165 | 0 |
| WEING | 8 | 8 | 0.0073 | 64 | 81 | 0.0010 | 78 |
| WEISH | 29 | 9 | 0.9951 | 20 | 89 | 0.6791 | 35 |

Tabela 4.3: Resultados das execuções para as instâncias do primeiro conjunto com 256 formigas em 8 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| Nome do conjunto de problemas | Número de problemas | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| HP | 2 | 6 | 1.7250 | 0 | 61 | 1.0457 | 0 |
| PB | 6 | 6 | 2.2536 | 13 | 65 | 1.2005 | 17 |
| PETERSEN | 6 | 6 | 0.4863 | 30 | 60 | 0.3120 | 34 |
| SENTO | 2 | 21 | 0.5424 | 0 | 219 | 0.2925 | 0 |
| WEING | 8 | 12 | 0.0029 | 69 | 123 | 0.0008 | 79 |
| WEISH | 29 | 13 | 0.8608 | 26 | 138 | 0.6264 | 39 |

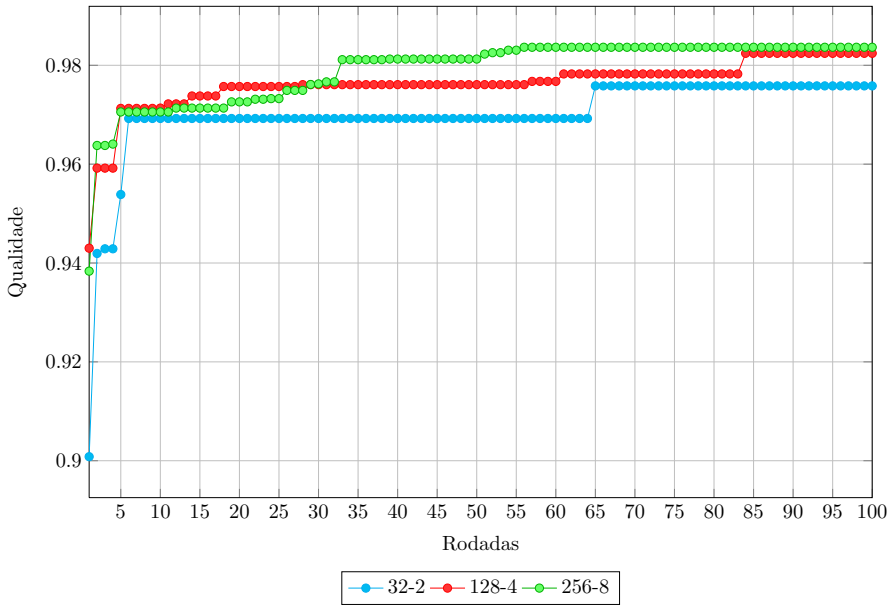



Figura 4.1: Gráfico de aproximação das soluções geradas à ótima em relação ao número de rodadas para a instância hp2 do MKP.

conjunto mais utilizado e possui instâncias mais difíceis. Posições das tabe-

Tabela 4.4: Resultados das execuções para as instâncias do segundo conjunto com 32 formigas em 2 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| | | | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|---|
| n | m | $\alpha$ | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| 100 | 5 | 0.25 | 7 | 1.3507 | 0 | 71 | 0.7219 | 3 |
| | | 0.50 | 9 | 0.6179 | 0 | 90 | 0.3204 | 1 |
| | | 0.75 | 11 | 0.2734 | 0 | 109 | 0.1305 | 5 |
| 250 | 5 | 0.25 | 25 | 0.7876 | 0 | 252 | 0.5503 | 0 |
| | | 0.50 | 33 | 0.3625 | 0 | 337 | 0.2585 | 0 |
| | | 0.75 | 41 | 0.1698 | 1 | 423 | 0.1102 | 0 |
| 500 | 5 | 0.25 | 104 | 0.4255 | 0 | 1013 | 0.3140 | 0 |
| | | 0.50 | 138 | 0.1865 | 0 | 1350 | 0.1394 | 0 |
| | | 0.75 | 178 | 0.1035 | 0 | 1694 | 0.0757 | 0 |
| 100 | 10 | 0.25 | 7 | 2.8539 | 0 | 75 | 2.0904 | 0 |
| | | 0.50 | 10 | 1.2618 | 0 | 101 | 0.9100 | 0 |
| | | 0.75 | 12 | 0.6209 | 0 | 126 | 0.4194 | 5 |
| 250 | 10 | 0.25 | 26 | 1.5187 | 0 | 263 | 1.1971 | 0 |
| | | 0.50 | 36 | 0.8320 | 0 | 368 | 0.6375 | 0 |
| | | 0.75 | 46 | 0.3516 | 0 | 470 | 0.2745 | 0 |
| 500 | 10 | 0.25 | 111 | 0.7634 | 0 | 1058 | 0.6229 | 0 |
| | | 0.50 | 153 | 0.3871 | 0 | 1473 | 0.3235 | 0 |
| | | 0.75 | 196 | 0.2265 | 0 | 1898 | 0.1762 | 0 |
| 100 | 30 | 0.25 | 9 | 4.0661 | 0 | 96 | 3.2013 | 0 |
| | | 0.50 | 14 | 2.4857 | 0 | 148 | 2.0043 | 0 |
| | | 0.75 | 20 | 1.1617 | 0 | 202 | 0.9161 | 0 |
| 250 | 30 | 0.25 | 32 | 3.3568 | 0 | 325 | 2.7859 | 0 |
| | | 0.50 | 49 | 1.7685 | 0 | 502 | 1.5224 | 0 |
| | | 0.75 | 67 | 0.8207 | 0 | 682 | 0.7148 | 0 |
| 500 | 30 | 0.25 | 145 | 2.2585 | 0 | 1473 | 2.0364 | 0 |
| | | 0.50 | 232 | 1.0761 | 0 | 2343 | 0.9530 | 0 |
| | | 0.75 | 320 | 0.6021 | 0 | 3209 | 0.5387 | 0 |

las que contém "-" são posições onde os dados não foram informados no trabalho da literatura.

Na coluna que representa o tempo consumido da Tabela 4.8 um fator de correção que leva em conta a frequência do processador foi considerado. Este fator é calculado através da divisão da frequência de clock do processador do trabalho dividido pela frequência do processador deste trabalho (3.3GHz). O tempo mostrado na tabela é a multiplicação do fator vezes o tempo original de cada trabalho. Este tempo com fator de correção representa uma aproximação do tempo de execução se o algoritmo fosse executado na mesma máquina que este trabalho, porém este fator não é completamente justo, pois a implementação em CUDA não utiliza o clock do processador principal, e não há como fazer uma estimativa da quantidade de tempo consumida por um código sequencial se fosse executado em uma

Tabela 4.5: Resultados das execuções para as instâncias do segundo conjunto com com 128 formigas em 4 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| | | | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|---|
| n | m | $\alpha$ | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| 100 | 5 | 0.25 | 17 | 1.0839 | 0 | 174 | 0.6104 | 6 |
| | | 0.50 | 24 | 0.4798 | 0 | 239 | 0.2313 | 5 |
| | | 0.75 | 30 | 0.2130 | 0 | 304 | 0.1185 | 14 |
| 250 | 5 | 0.25 | 88 | 0.6766 | 0 | 898 | 0.4617 | 0 |
| | | 0.50 | 126 | 0.3192 | 0 | 1277 | 0.2238 | 0 |
| | | 0.75 | 164 | 0.1496 | 0 | 1656 | 0.0940 | 0 |
| 500 | 5 | 0.25 | 308 | 0.3664 | 0 | 3101 | 0.2883 | 0 |
| | | 0.50 | 439 | 0.1659 | 0 | 4409 | 0.1257 | 0 |
| | | 0.75 | 567 | 0.0921 | 0 | 5737 | 0.0670 | 0 |
| 100 | 10 | 0.25 | 17 | 2.6951 | 0 | 179 | 2.0092 | 0 |
| | | 0.50 | 25 | 1.2395 | 0 | 258 | 0.9198 | 0 |
| | | 0.75 | 33 | 0.5866 | 1 | 334 | 0.4281 | 6 |
| 250 | 10 | 0.25 | 92 | 1.4070 | 0 | 943 | 1.1747 | 0 |
| | | 0.50 | 138 | 0.7796 | 0 | 1402 | 0.6298 | 0 |
| | | 0.75 | 184 | 0.3253 | 0 | 1862 | 0.2514 | 0 |
| 500 | 10 | 0.25 | 323 | 0.7062 | 0 | 3262 | 0.5699 | 0 |
| | | 0.50 | 485 | 0.3680 | 0 | 4856 | 0.3039 | 0 |
| | | 0.75 | 647 | 0.2079 | 0 | 6463 | 0.1713 | 0 |
| 100 | 30 | 0.25 | 22 | 3.9973 | 0 | 223 | 3.1965 | 0 |
| | | 0.50 | 36 | 2.5919 | 0 | 361 | 2.1541 | 0 |
| | | 0.75 | 50 | 1.1659 | 0 | 507 | 0.9684 | 0 |
| 250 | 30 | 0.25 | 113 | 3.3037 | 0 | 1148 | 2.9607 | 0 |
| | | 0.50 | 185 | 1.8060 | 0 | 1873 | 1.6096 | 0 |
| | | 0.75 | 261 | 0.8503 | 0 | 2627 | 0.7522 | 0 |
| 500 | 30 | 0.25 | 406 | 2.2872 | 0 | 4088 | 2.0467 | 0 |
| | | 0.50 | 667 | 1.0776 | 0 | 6689 | 0.9784 | 0 |
| | | 0.75 | 928 | 0.6165 | 0 | 9283 | 0.5539 | 0 |

*GPGPU*, pois isto depende da implementação e das características do algoritmo.

Os resultados de [92] não são mostrados nas tabelas pois o trabalho implementa um método exato de *branch-and-bound* com busca de decisão (*resolution search*), logo o tempo de execução é centenas de vezes maior para qualquer instância, visto que o algoritmo deve encontrar uma solução que assume ser a ótima e provar ser mesmo a ótima. Por exemplo, para as instâncias de tamanho $250 \times 10$, o algoritmo consome de 1 a 2000 segundos para encontrar uma solução e mais de 1995 a 34976 segundos para a prova, consumindo aproximadamente 10 horas no pior caso e 0.5 horas no melhor caso.

Em [7] um método de programação dinâmica híbrida(HDP) com uma melhoria no cálculo de limitantes inferiores ou *lower bounds* (LBC) foi implementado. Considerando a tabela com o fator de correção, nota-se que o HDP+LBC obtêm melhores resultados para todas as instâncias, com exceção

Tabela 4.6: Resultados das execuções para as instâncias do segundo conjunto com com 256 formigas em 8 colônias. O tempo apresentado é a média de 10 execuções para cada instância.

| | | | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|---|
| n | m | $\alpha$ | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| 100 | 5 | 0.25 | 26 | 0.9442 | 0 | 267 | 0.5119 | 8 |
| | | 0.50 | 36 | 0.3980 | 1 | 365 | 0.1915 | 5 |
| | | 0.75 | 46 | 0.1775 | 3 | 461 | 0.0903 | 18 |
| 250 | 5 | 0.25 | 149 | 0.6090 | 0 | 1500 | 0.4369 | 0 |
| | | 0.50 | 212 | 0.2872 | 0 | 2135 | 0.1974 | 0 |
| | | 0.75 | 277 | 0.1368 | 0 | 2787 | 0.0816 | 0 |
| 500 | 5 | 0.25 | 557 | 0.3450 | 0 | 5598 | 0.2664 | 0 |
| | | 0.50 | 787 | 0.1517 | 0 | 7909 | 0.1164 | 0 |
| | | 0.75 | 1020 | 0.0824 | 0 | 10225 | 0.0635 | 0 |
| 100 | 10 | 0.25 | 28 | 2.5208 | 0 | 284 | 1.8316 | 0 |
| | | 0.50 | 41 | 1.0806 | 0 | 415 | 0.8114 | 0 |
| | | 0.75 | 53 | 0.5413 | 3 | 540 | 0.3838 | 10 |
| 250 | 10 | 0.25 | 156 | 1.3337 | 0 | 1571 | 1.0753 | 0 |
| | | 0.50 | 235 | 0.7251 | 0 | 2364 | 0.5811 | 0 |
| | | 0.75 | 314 | 0.3129 | 0 | 3157 | 0.2388 | 0 |
| 500 | 10 | 0.25 | 589 | 0.6553 | 0 | 5920 | 0.5355 | 0 |
| | | 0.50 | 876 | 0.3482 | 0 | 8790 | 0.2865 | 0 |
| | | 0.75 | 1162 | 0.1995 | 0 | 11673 | 0.1580 | 0 |
| 100 | 30 | 0.25 | 35 | 3.7178 | 0 | 356 | 3.0288 | 0 |
| | | 0.50 | 58 | 2.4294 | 0 | 583 | 2.0567 | 0 |
| | | 0.75 | 82 | 1.1367 | 0 | 824 | 0.9297 | 0 |
| 250 | 30 | 0.25 | 193 | 3.2283 | 0 | 1943 | 2.8289 | 0 |
| | | 0.50 | 320 | 1.7218 | 0 | 3218 | 1.5595 | 0 |
| | | 0.75 | 450 | 0.7946 | 0 | 4527 | 0.7258 | 0 |
| 500 | 30 | 0.25 | 736 | 2.1792 | 0 | 7401 | 1.9765 | 0 |
| | | 0.50 | 1203 | 1.0448 | 0 | 12059 | 0.9600 | 0 |
| | | 0.75 | 1666 | 0.5971 | 0 | 16730 | 0.5325 | 0 |

Tabela 4.7: Tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura para o segundo conjunto.

| | | 256-8 | | HDP+LBC [7] | | Kernel Search [91] | | NP(R) [93] | |
|---|---|---|---|---|---|---|---|---|---|
| n | m | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| 100 | 5 | 364 | 0.2645 | 600 | 0.57 | - | - | - | 0.53 |
| 250 | 5 | 2140 | 0.2386 | 1000 | 0.16 | 196000 | 0.0 | - | 0.24 |
| 500 | 5 | 7910 | 0.1487 | 1130 | 0.07 | 1088000 | 0.002 | - | 0.08 |
| 100 | 10 | 413 | 1.0089 | 1000 | 0.95 | - | - | - | 1.10 |
| 250 | 10 | 2364 | 0.6317 | 970 | 0.32 | 1465000 | 0.001 | - | 0.48 |
| 500 | 10 | 8794 | 0.3266 | 4030 | 0.16 | 1579000 | 0.019 | - | 0.19 |
| 100 | 30 | 587 | 2.005 | 3970 | 1.81 | - | - | - | 1.45 |
| 250 | 30 | 3229 | 1.7047 | 21200 | 0.77 | 2189000 | 0.013 | - | 0.80 |
| 500 | 30 | 12063 | 1.1563 | 93370 | 0.42 | 2100000 | 0.062 | - | 0.49 |

Tabela 4.8: Tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura para o segundo conjunto com fator de correção no tempo de execução.

| | | 256-8 | | HDP+LBC [7] | | Kernel Search [91] | | NP(R) [93] | |
|---|---|---|---|---|---|---|---|---|---|
| n | m | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| 100 | 5 | 364 | 0.2645 | 291 | 0.57 | - | - | - | 0.53 |
| 250 | 5 | 2140 | 0.2386 | 484 | 0.16 | 166286 | 0.0 | - | 0.24 |
| 500 | 5 | 7910 | 0.1487 | 548 | 0.07 | 923059 | 0.002 | - | 0.08 |
| 100 | 10 | 413 | 1.0089 | 485 | 0.95 | - | - | - | 1.10 |
| 250 | 10 | 2364 | 0.6317 | 470 | 0.32 | 1242906 | 0.001 | - | 0.48 |
| 500 | 10 | 8794 | 0.3266 | 1954 | 0.16 | 1339623 | 0.019 | - | 0.19 |
| 100 | 30 | 587 | 2.005 | 1925 | 1.81 | - | - | - | 1.45 |
| 250 | 30 | 3229 | 1.7047 | 10278 | 0.77 | 1857147 | 0.013 | - | 0.80 |
| 500 | 30 | 12063 | 1.1563 | 45266 | 0.42 | 1781640 | 0.062 | - | 0.49 |

da menor ($100 \times 5$), onde a diferença de qualidade das soluções é notável (0.26 contra 0.57), porém a diferença de tempo é muito pequena.

Um algoritmo de *kernel search* (KS) foi implementado em [91]. Comparando os resultados obtidos nota-se que a qualidade das soluções encontradas pelo KS é extremamente alta, 0.062 no pior caso que são as maiores instâncias, que tem tamanho $500 \times 30$, porém o tempo consumido também é alto, em torno de 148 vezes maior do que o ACO para estas maiores instâncias e até 575 vezes maior para as instâncias de tamanho $250 \times 30$. Portanto o KS encontra soluções muito boas e próximas da ótima para todas as instâncias executadas, porém o tempo consumido é muito alto.

Em [93] vários métodos heurísticos são comparados e um novo método utilizando redução de problemas (NP(R)) é implementado. Entre as heurísticas comparadas está o HDP de [7] e um algoritmo genético (GA) [88]. Neste trabalho nenhum tempo de execução foi informado, logo apenas a qualidade das soluções será comparada. O ACO encontrou soluções melhores para apenas três tamanhos de instâncias: $100 \times 5$, $250 \times 5$ e $100 \times 10$, porém como o tempo não é mostrado, não se pode concluir qual algoritmo é melhor para cada caso.

## 4.2 MOCHILA BINÁRIA OU 0-1

A variação mais comum do problema da mochila é o problema da mochila binária, que restringe o número de cópias dentro da mochila para zero ou um (escolhido ou não escolhido) em apenas uma dimensão. Este problema é igual ao MKP, porém o número de dimensões é fixo em um. Este problema pode ser formulado matematicamente como: seja $w_j > 0$ ser o peso e $p_j > 0$ o valor do objeto j, com j pertencendo ao conjunto de objetos N e $|N|$ sendo igual a $n$, se este objeto está na mochila $x_j = 1$, senão $x_j = 0$. A capacidade máxima da mochila é $W$. Deve-se maximizar a soma de $p_j x_j$ para todo j, assim maximizando o valor da nossa mochila. Preparando-se para uma fórmula recursiva, pode-se dizer que a solução ótima desta mochila é $F(W, N)$. 4.11 mostra a formulação matemática da mochila binária.

$$\begin{aligned} F(C, n) &= \max \sum_{j=1}^{n} p_j x_j (p_j \in \mathbb{N}^*) \\ \text{sujeito a} \sum_{j=1}^{n} w_j x_j &\leqslant C (w_j, C \in \mathbb{N}^*) \end{aligned} \tag{4.11}$$

Como o problema parece simples, tende-se a imaginar que a solução também é simples, porém isto não é verdade. Um algoritmo ingênuo que tentar gerar todas as soluções possíveis, terá que gerar $2^n$ soluções, onde $n$ é o número de objetos. Este problema é um problema difícil, tanto é que pertence a classe de problemas NP-difíceis, portanto não existe um algoritmo que solucione o problema em tempo polinomial.

Mesmo não tento um algoritmo polinomial para encontrar a solução ótima, existe um que consegue fazê-lo rapidamente usando programação dinâmica. Porém existe um problema, este algoritmo é pseudo-polinomial, o que significa que ele não depende apenas do tamanho da entrada (neste caso, o número de objetos), ele depende também da capacidade máxima da mochila. Portanto para uma certa instância com uma capacidade máxima muito grande, este algoritmo pode ter uma performance ruim. Como o algoritmo utiliza programação dinâmica, uma subestrutura ótima deve existir no problema, e ela é simples: dados $n$ objetos, se o n-ésimo objeto está na mochila, a solução ótima é o valor do objeto n-ésimo, mais a solução ótima da mochila que não tem o n-ésimo objeto e cuja capacidade máxima é $W$ menos o peso do n-ésimo objeto. Se o n-ésimo objeto não está na mochila, a solução ótima do problema é a solução ótima do problema cuja mochila tem a mesma capacidade máxima, porém com um objeto a considerar a menos: o n-ésimo objeto. Matematicamente:

$$F(W, N) = \begin{cases} F(W, N - i) \text{ se o objeto } i \\ \text{não cabe ou não está na mochila.} \\ F(W - w_i, N - i) + p_i \text{ se} \\ \text{o objeto } i \text{ está na mochila.} \end{cases} \tag{4.12}$$

Como cada peso máximo de $0$ até $W$ pode ser usado, e o número de objetos varia de $0$ até $n$, uma matriz é utilizada para representar este algoritmo, chamada de $M$. Esta matriz possui $n$ linhas, cada linha representando um objeto, e $W$ colunas, representando cada possível capacidade de mochila. Para cada posição $i, w$ da matriz, $M[i][w]$ contém a solução ótima para a mochila cuja capacidade máxima é $w$ e apenas os $i$ primeiros objetos são considerados. É fácil de ver que quando $i = 0$ (não existem objetos) e quando $w = 0$ (não cabe nenhum objeto na mochila) a solução deve ser $0$. Pode-se dizer que a solução quando $w = 0$ é $0$, apenas se nenhum objeto pesa o, o que é plausível, visto que se um objeto tem peso $0$, ele estará em qualquer mochila ótima e pode ser ignorado, já que o valor de um objeto nunca é negativo. Agora a matriz é preenchida, resolvendo cada subproblema:

$$M[i][w] = \begin{cases} max(M[i-1][w], \\ p_i + M[i-1][w-w_i]) & \text{se } w_i \leqslant w \\ M[i-1][w] & \text{se } w_i > w \end{cases} \tag{4.13}$$

Cada posição desta matriz é preenchida apenas uma vez, e apenas duas outras posições são consultadas a cada preenchimento, assim a complexidade é $O(nW)$. Se $W$ é muito grande, pode-se apenas calcular as posições da matriz que serão usadas. Pode ser demonstrado que este algoritmo, com poucas mudanças, independente de $W$, pode resolver o problema preenchendo $O(2^n)$ posições. Portanto é possível encontrar a solução ótima de uma mochila binária em tempo $O(min(nW, 2^n))$.

O algoritmo paralelo de programação dinâmica para este problema foi baseado no algoritmo proposto por Boyer, El Baz e Elkihel [94].

### 4.2.1 *Colônia de formigas para o Problema da Mochila Binária*

O algoritmo de colônia de formigas para o problema da mochila binária é igual ao do algoritmo para o MKP, com apenas uma diferença, o número de dimensões é fixo em um ($m = 1$).

Quanto ao algoritmo de programação dinâmica, note que cada posição da matriz em uma mesma linha é independente: elas não dependem do valor de alguma posição da mesma linha. Como elas são independentes, é possível calcular todas as colunas de uma linha em paralela, e é exatamente isso que o código em CUDA faz. Dada uma instância do problema, são lançadas *threads* suficientes para cobrir todas as colunas, e então preenchidas as $n$ linhas com $n$ lançamentos de *kernel*, cada *kernel* preenchendo uma linha.

Se após a execução do algoritmo deseja-se saber quais objetos foram escolhidos, a matriz gerada é verificada e alguns cálculos feitos na direção contrária. Mas se apenas o valor da solução ótima é suficiente. Neste caso, é possível reduzir o tamanho da matriz de $nW$ para $2W$. Isto é feito através do não armazenamento da matriz inteira, mas apenas de duas linhas, pois note que para calcular as posições de uma linha, apenas a linha exatamente acima dela é usada, portanto é suficiente armazenar apenas duas linhas na memória, a linha acima e a linha que está sendo calculada. Se 1280MB de memória estiverem disponíveis, pode-se armazenar 335.544.320 inteiros de 4-*bytes*. Dividindo em duas linhas, cada linha pode possuir 167.772.160 inteiros, e neste cálculo é ignorando valores necessários para o algoritmo, como o peso e o valor dos objetos e a memória local usada por cada *thread*. Este cálculo foi feito para mostrar que existe um limite para nosso algoritmo: existe um certo valor de $W$, independente do tamanho da entrada $n$, que vai fazer com que o uso de memória ultrapasse a disponível. Se for necessário obter não só o valor da solução como os objetos, este limite é menor ainda, pois a matriz possui mais de 2 linhas.

Uma *thread* é lançada para cada coluna da matriz, mas este problema não é levado em consideração, visto que o problema de falta de memória ocorreria primeiro. Em uma GPU com *compute capability* 2.0, o limite de *threads*

lançadas por *kernel* é $65535 * 65535 * 65535 * 1024$. 65535 é o número máximo de blocos para uma dimensão do *grid*, e pode ter três dimensões. Cada bloco pode conter no máximo 1024. O número total de *threads* que podem ser lançadas é $288217182 \times 10^17$. Este cálculo é inteiramente teórico, se cada *thread* necessitasse de um mínimo de memória local, a memória ocupada por elas ultrapassaria a memória disponível.

### 4.2.2 *Resultados experimentais*

As instâncias utilizadas para os testes da mochila binária usadas foram criadas por um gerador pré-existente, desenvolvido por Pisinger [95], disponível em [96]. Quatro tamanhos foram escolhidos para os testes, o número de objetos foi entre 512 e 16384 e o peso máximo para cada número objetos foi entre 2048 e 65536, o número de objetos e o peso máximo de cada instância pode ser visto na Tabela 4.9.

O conjunto de configurações utilizado foi o mesmo utilizado na execução do MKP. O número de formigas pelo número de colônias variado entre 8/1, 32/2, 128/4. Os parâmetros $\alpha$ e $\beta$ testados com 4 e 1, 1 e 4 , 1 e 1. $t_{min}$ e $t_{max}$ iguais a 0.01 e 6, respectivamente. E o fator de evaporação variado entre 0.1, 0.3 e 0.7.

Além dos testes de qualidade de solução sobre um certo número de rodadas, um teste em cada instância foi executado até que fosse encontrada uma solução com diferença máxima de 0.05%, ou seja, até encontrar uma solução com qualidade igual ou superior a 0.95.

Porém, devido à maneira como as instâncias são geradas [95], o número muito grande de objetos, cada um com peso entre zero e o peso máximo, o algoritmo otimização de formigas não foi capaz de encontrar soluções próximas à ótima em nenhum dos testes, nem mesmo após 10.000 rodadas. Nenhum conjunto de instâncias como existe a biblioteca OR para o MKP foi encontrado, então apenas estas soluções geradas foram testadas. Como as qualidades das soluções encontradas foram abaixo de 0.2 e os tempos de execução altos devido ao número de objetos, os resultados foram omitidos por não serem úteis.

Pode-se considerar os resultados da mochila multidimensional, pois, é possível notar que na Tabela 4.1 a diferença de tempo entre instâncias com $n$ e $\alpha$ igual, porém com $m$ diferente é pequena, assim, pode-se assumir que o tempo de execução médio do algoritmo para a mochila binária é muito parecido com o da mochila multidimensional, visto que o número de dimensões não é determinante no tempo de execução total.

Levando em conta apenas os algoritmos de programação dinâmica, nota-se que a implementação em GPGPU resultou em um *speedup* de até 60 na maior instância.

Na Figura 4.2 pode-se notar que, apesar de o tempo crescer conforme o tamanho e o peso máximo de cada instância aumenta, o tempo de execução do algoritmo paralelo mantêm-se quase linear em relação ao crescimento do algoritmo sequencial.

Tabela 4.9: Resultados das execuções para o problema da mochila binária com programação dinâmica.

| Tamanho do problema (número de objetos por peso máximo da mochila) | Programação Dinâmica sequencial (ms) | Programação Dinâmica em CUDA (ms) |
|---|---|---|
| 512/2048 | 52 | 6 |
| 1024/4096 | 213 | 10 |
| 2048/8192 | 847 | 25 |
| 4096/16384 | 3377 | 73 |
| 8192/32768 | 13469 | 248 |
| 16384/65536 | 53994 | 874 |

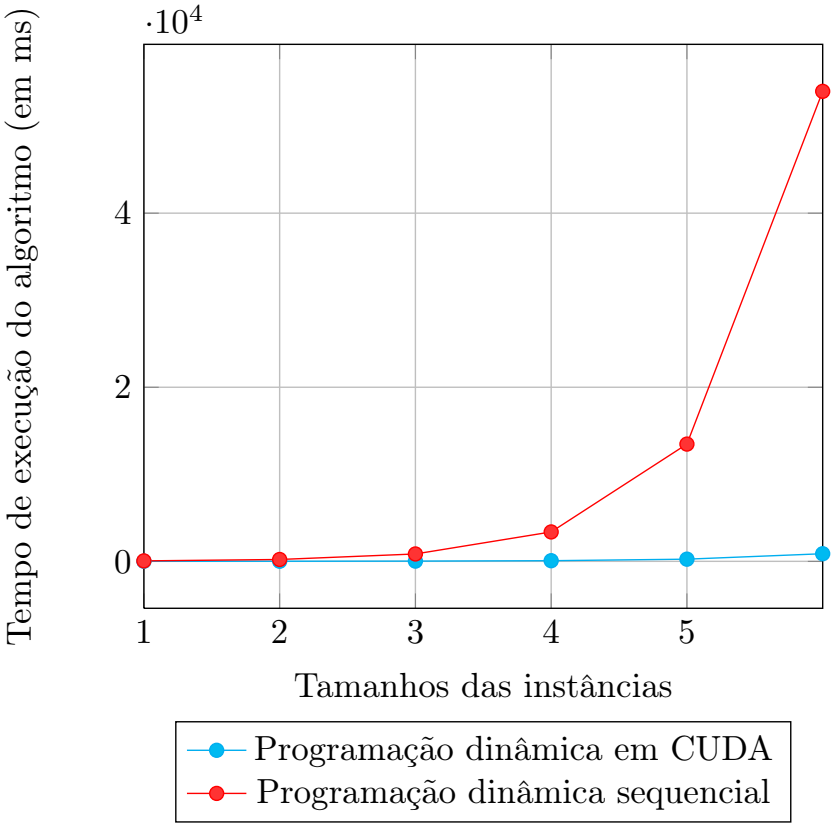



Figura 4.2: Crescimento dos tempos dos algoritmos sequenciais e paralelos em relação ao tamanho das instâncias.

# 5

# PROBLEMA QUADRÁTICO DE ALOCAÇÃO

O problema quadrático de alocação, ou *quadratic assignment problem* (*QAP*), é um problema de otimização combinatória que pode ser visto como a tarefa de atribuir fábricas à locais levando em conta o fluxo entre as fábricas e as distâncias entre os locais. Se existem $n$ fábricas e $n$ locais, a solução é um vetor cuja posição representa um local e o valor desta posição a fábrica que foi colocada no local. Um par de fábricas que tem um fluxo alto de materiais entre elas devem ser colocadas próximas uma da outra, minimizando os custos de transporte. Os dados são representados por três tabelas $n \times n$: a matriz A representa a distância entre os locais, uma posição $i, j$ ($a_{ij}$) desta matriz representa a distância entre os locais $i$ e $j$. Similarmente, a matriz B representa o fluxo de materiais entre as fábricas. A matriz C representa o custo específico de instalar uma fábrica em um local, portanto uma posição $c_{ij}$ representa o custo de se instalar a fábrica j no local i. A remoção da matriz C do problema não modifica a complexidade, portanto é desconsiderada na maioria dos problemas.

Matematicamente, dadas três matrizes $n \times n$, A, B e C, o objetivo é encontrar uma permutação $\pi$ que minimize a função:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} \times b_{\pi_i \pi_j} + \sum_{i=1}^{n} c_{i\pi_i} \tag{5.1}$$

Esta formulação é chamada de versão de Koopmans-Beckmann [97]. Se as três matrizes são dadas, esta instância do problema é denotada como *QAP*(A, B, C). Se apenas A e B são dados, ou seja, não existe custo específico de instalação de uma fábrica em um local ou este custo é constante para todas as fábricas em qualquer local, o problema pode ser denotado como *QAP*(A, B). Esta formulação é a mais utilizada, e será utilizada também no desenvolvimento deste trabalho.

Um exemplo é expresso pelas Figuras 5.1 e 5.2 e as matrizes de fluxo A e de distância B correspondentes.

$$A = \begin{pmatrix} 0 & 7 & 2 \\ 7 & 0 & 4 \\ 2 & 4 & 0 \end{pmatrix} \text{ e } B = \begin{pmatrix} 0 & 2 & 4 \\ 2 & 0 & 3 \\ 4 & 3 & 0 \end{pmatrix}$$

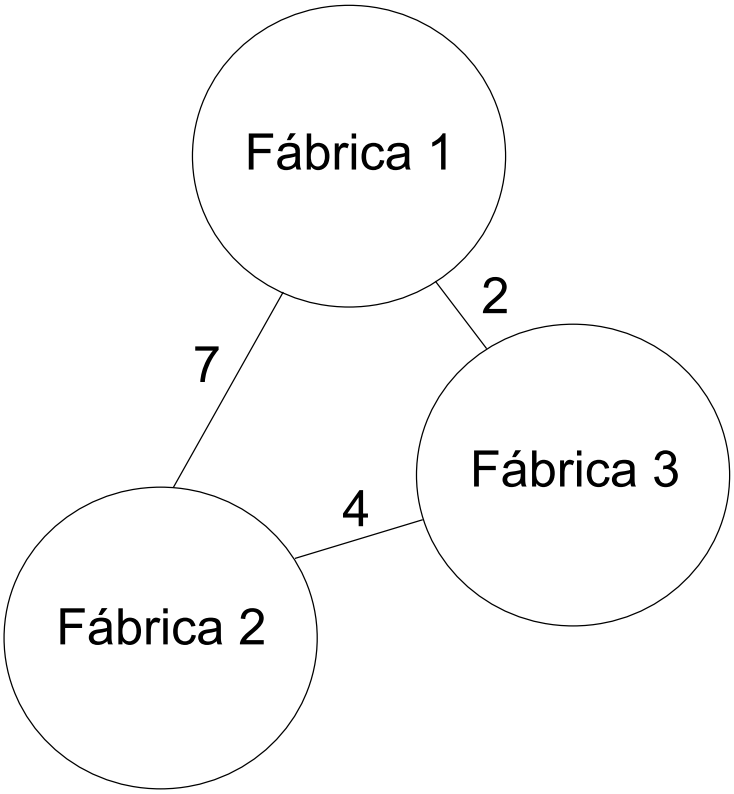



Figura 5.1: As fábricas e a quantidade de material que deve ser transportada entre elas.

Uma possível solução para o problema descrito nas Figuras de 1 e 2, seria a Fábrica 1 no Local 1, Fábrica 2 no Local 2 e Fábrica 3 no Local 3: $\pi = (1, 2, 3)$ . O resultado de acordo com a Equação 5.1 seria:

$$((7 \times 2) + (2 \times 4)) + ((7 \times 2) + (4 \times 3)) + ((2 \times 4) + (4 \times 3)) = 68$$

Várias pesquisas sobre o *QAP* foram realizadas nas últimas três décadas, e o *QAP* continua sendo um dos problemas de otimização mais difíceis de se resolver. Nenhum algoritmo exato consegue resolver problemas de tamanho maior que vinte ($n > 20$) em tempo razoável, já que existem 20! soluções possíveis em um problema de tamanho 20.

Sahni e Gonzalez [98] provaram que o *QAP* é NP-difícil e que, até mesmo para encontrar uma solução aproximada, com um erro máximo constante distante da solução ótima, não pode ser feito em tempo polinomial, a não ser que $P = NP$. A prova da inclusão do *QAP* na classe NP-difícil utiliza uma redução a partir do problema do ciclo hamiltoniano em grafos, conforme Sahni e Gonzalez [98]. Até o momento, apenas para um caso muito específico do *QAP*, chamado Problema do Arranjo Linear Denso, existe uma

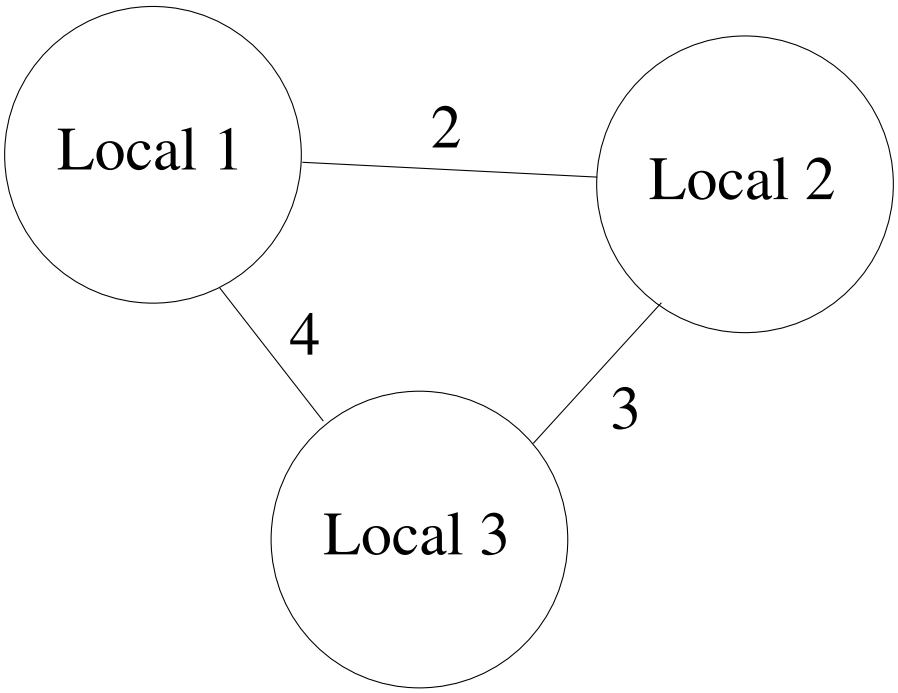



Figura 5.2: Os locais e as distâncias entre eles.

técnica de aproximação de tempo polinomial, descrita por Arora, Frieze e Kaplan [99].

Existem outras formulações do *QAP*, que são matematicamente iguais, mas que exploram diferenças nas características estruturais do problema. Estas diferenças podem levar a métodos alternativos de aproximação. Alguns exemplos de formulações são:

- Formulação quadrática de inteiros [1]
- Formulação utilizando traços de matrizes [1]
- Produto de Kronecker [1]
- Formulação de Lawler [1]

Depois de entender o *QAP*, é fácil encontrar uma aplicação para ele na vida real. Nas próximas subseções algumas aplicações do *QAP* são descritas.

## 5.1 APLICAÇÕES PRÁTICAS

### 5.1.1 *Organização de um Hospital (Hospital Layout)*

Esboçar o projeto de um hospital, colocando cada um de seus setores em um local, é uma tarefa fácil, porém alcançar uma organização perfeita, em que a distância total que cada paciente percorre por ano, é um problema complexo.

Em 1975, Alwalid N. Elshafei [12], do Instituto de Planejamento do Cairo, Egito, investigou a organização ótima de clínicas (como salas de Raio-X,

Emergências, Ortopedia e outras) em um hospital e definiu então, como sendo a colocação (ou atribuição) ótima, a que minimiza a distância total de deslocamento de pacientes entre as clínicas, medidos em metros-paciente por ano (mpa). Assim, fica fácil entender porque na maioria dos hospitais do mundo, o setor de emergências se localiza na parte da frente do hospital, pois, assim, pacientes que chegam ao hospital necessitando de ajuda urgente precisam se locomover o mínimo possível para serem atendidos. Elshafey ainda, neste mesmo estudo, focou em diminuir o problema de superlotação de um grande hospital no Cairo. Este hospital possuía 17 clínicas e tinha mais de 700 pacientes por dia. O grande número de pacientes somados à má colocação das clínicas causavam um tráfego de pacientes enorme e grandes atrasos em várias operações no hospital.

Elshafei e Bazaraa desenvolveram um algoritmo heurístico (E & B) em duas fases para resolver esta instância do *QAP*. A primeira determinava uma solução inicial e a segunda melhorava esta solução. Além das 17 salas, o departamento possuía uma sala de recebimento e uma sala de gravação. A sala de gravação não possui tráfego de pacientes, logo, sua colocação não faz parte da solução, restando 18 clínicas. Apenas uma das 18 salas precisava do dobro de espaço, logo, a esta clínica, duas salas adjacentes seriam escolhidas. O problema então era atribuir 19 clínicas a 19 salas.

A implementação da heurística gerava um padrão e combinava elementos um a um sucessivamente, e armazenava as soluções em uma lista ordenada de acordo com a métrica $\mathrm{mpa}$. O processamento parou quando nenhuma solução melhor foi encontrada após 50 diferentes padrões.

Inicialmente, o hospital possuía $13,973,298\mathrm{mpa}$. A heurística aplicada a este problema encontrou uma solução de $11,281,298\mathrm{mpa}$. Esta otimização eliminou mais de 20% de tráfego desnecessário de pacientes, resultando em um centro de tratamento mais eficiente, através da implementação de um procedimento que demorou menos de dois minutos (136s utilizando um IBM 360/40) de execução. Esta organização encontrada foi então implantada no hospital em questão.

Outra aplicação equivalente foi utilizada por Krarup e Pruzan [100] para organizar a clínica *Klinikum Regensburg* na Alemanha.

### 5.1.2 *Colocação de Componentes Eletrônicos Interligados em Placas de Circuito Impresso ou Microchip*

A colocação de componentes em uma placa de circuito impresso, se feita de uma boa maneira, pode reduzir o tempo de atraso de sinais de um componente a outro, melhorando seu desempenho geral e diminuindo os custos ao fabricante, já que uma distância menor de conexões será utilizada. Este problema foi proposto inicialmente em 1961 por Steinberg [13], e mais tarde, este problema recebeu seu nome: "*Steinberg Wiring Problem*".

Em seu trabalho, Steinberg tentou colocar de maneira ótima 34 componentes em uma placa com 36 posições, e estes componentes tinham um total de 2625 interconexões. Dois componentes artificiais que não tinham cone-

xões com nenhum outro foram criados para preencher a placa. Até o ano de 2001, o método de Busca Tabu (*Taboo Search*) [101] tinha produzido o melhor resultado para o problema original, e tinha valor igual a 9526. Ainda em 2001, Brixius e Anstreicher [102] implementaram um algoritmo *branch and bound* utilizando o cálculo de limitante de Gilmore-Lawler [97] para resolver este problema. A árvore de soluções utilizou aproximadamente $7.75 \times 10^8$ nós e demorou aproximadamente 186 horas de processamento em um Pentium III 800Mhz. Esta execução então, concluiu que o valor 9526 é, de fato, a melhor solução possível para o *SWP*. Brixius e Anstreicher ainda comentaram que uma melhoria no tempo total na solução do problema poderia ser encontrada.

### 5.1.3 *Problemas que podem ser reduzidos ao QAP [1]*

- Particionamento de grafos e clique máximo.
- Problema do caixeiro viajante.
- Problema do arranjo linear.
- Problema de empacotamento em grafos.

Após esta introdução ao problema, o algoritmo de otimização de colônia de formigas utilizado para resolver este problema, uma variação paralela do HAS-QAP [103], é apresentado na seção abaixo. Os resultados obtidos e comparações com outros trabalhos são apresentados na seção seguinte.

## 5.2 COLÔNIA DE FORMIGAS PARA O QAP

Para que a otimização de colônias de formigas seja utilizada para solucionas o *QAP*, é preciso determinar o que é uma solução e o que representa a tabela de feromônios.

No *QAP*, uma solução é uma permutação de $n$ elementos, de 0 até $n-1$ e um valor, calculado utilizando as duas tabelas do problema (de distância e de fluxo). Assim, é sabido que cada formiga de um sistema aplicado ao *QAP*, utilizará no mínimo memória equivalente a $O(n)$. Ainda, o espaço ocupado pelas duas matrizes do problema é comum a todas as formigas e colônias, sendo proporcional a $O(n^2)$.

A tabela de feromônios, quando utilizada no *QAP* possui dados de o quanto uma fábrica é uma boa candidata a todos os locais. Um elemento $t_{ij}$ da tabela, indica quanto a fábrica $i$, se colocada no local $j$, é positiva à solução. Quanto maior este valor é, maior é a probabilidade de a fábrica $i$ estar no local $j$. Como a trilha é única a cada colônia de formigas, a memória ocupada por cada colônia é proporcional a $O(n^2)$.

Existem diversas maneiras de implementar um algoritmo de formigas. É possível construir soluções ou modificá-las, utilizar passos globais para modificar a maneira como as formigas trabalham, entre outros. Para o QAP, o HAS [103] foi escolhido para ser implementado por ser um algoritmo que já

obteve bons resultados. Este algoritmo e seus detalhes são apresentados na subseção

### 5.2.1 *Hybrid Ant System - Quadratic Assignment Problem (HAS-QAP)*

O *HAS-QAP*, proposto por Gambardella, Taillard e Dorigo [103], é um sistema de colônia de formigas que, diferentemente dos outros, gera uma nova solução modificando a solução anterior, usando a tabela de feromônios.

Nesta variação de *ACO*, apenas um passo global de atualização da tabela de feromônios é feita. Isto faz com que a busca seja mais agressiva e que menos tempo seja consumido para que boas soluções sejam encontradas. Além desta mudança, um passo global de intensificação é realizado para diminuir ainda mais o tempo para encontrar boas soluções. Porém estes passos de busca agressiva podem fazer o sistema convergir a uma solução precoce do algoritmo, evitando a busca de possíveis melhorias. Para corrigir este problema, outro passo global é realizado periodicamente, um passo de diversificação, que apaga todas as trilhas de feromônio adicionadas.

Cada formiga, como dito anteriormente, não cria uma nova solução a cada passo, mas modifica a atual, utilizando apenas a tabela de feromônio como informação. Após esta modificação, uma fase de melhoria é executada. Esta fase realiza uma rápida busca local e considera apenas o objetivo principal. Após a geração de todas as soluções existe o problema de escolher qual é a próxima a cada iteração. Para resolver estes problemas os mecanismos de intensificação e diversificação existem. A intensificação, quando ativa, faz com que a formiga, após gerar sua solução, volte a solução inicial se esta é melhor do que a gerada. Caso contrário continua onde está. O passo de diversificação reinicia o algoritmo quando as soluções não melhoram. Este passo consiste em reinicializar a trilha de feromônio e as soluções geradas associadas às formigas.

A cada rodada, todas as formigas modificam a solução anterior realizando $R$ trocas de posições na sua permutação, onde $R$ é um parâmetro passado para o algoritmo. Estas trocas são divididas em dois procedimentos de escolha de índices a serem trocados: O primeiro índice para a troca é sempre aleatório. O segundo pode ser *exploitation* ou *exploration*. O procedimento *exploitation* é escolhido com probabilidade $q$, e escolhe o índice a ser trocado de maneira gulosa utilizando a tabela de feromônios, ou seja, escolhe sempre a melhor fábrica a ser colocada na posição gerada aleatoriamente anteriormente. Já o procedimento de *exploration*, escolhido com probabilidade $1 - q$, escolhe probabilisticamente qual fábrica será escolhida, utilizando a tabela de feromônios. Fábricas que, de acordo com a tabela são melhores candidatas ao local gerado, têm maior probabilidade de serem escolhidas.

A tabela de feromônios é atualizada utilizando apenas a melhor solução gerada até o momento. Primeiramente ocorre uma evaporação de $q_1$ de toda a tabela. Se $q_1$ é igual a $1$, toda a trilha evapora, se é igual a $0$, não há evaporação. Então, as posições da melhor permutação até o momento são reforçadas de acordo com um parâmetro $q_2$ utilizando como quantificador

de feromônio o valor desta permutação. Se a solução não melhora por S rodadas, a trilha é apagada, reinicializada e reforçada utilizando a melhor solução, para que não se perca tudo que já foi gerado.

A busca local implementada por [103] funciona da seguinte maneira: dada uma formiga, após gerar a sua solução, um vetor aleatório $v$ com elementos de 1 até n é gerado. Para cada posição deste vetor um outro vetor $w$ com as mesmas características é gerado. Então, dado um valor $v_i$ do vetor $v$, para cada valor $w_j$ do vetor $w$, trocam-se os valores das posições $v_i$ e $w_j$ da permutação da solução gerada pela formiga. Após a troca, verifica-se se o valor da solução melhorou, se não melhorou a troca é desfeita, caso contrário a permutação modificada é mantida.

O total de trocas feitas é $n^2$, porém, para cada troca, deve-se calcular o valor da solução modificada. A comparação e a escolha entre manter ou não a troca é feita em um número constante de passos. O cálculo do valor de uma solução dada a permutação é utiliza $O(n^2)$, portanto a complexidade da busca local é $O(n^4)$. Porém em [103] existe um método matemático que, dada uma permutação e dois valores que são as posições que serão trocadas, calcula a diferença entre os valores das soluções com e sem troca em $O(n)$. Portanto a complexidade da busca local que inicialmente era $O(n^4)$ pode ser melhorada para $O(n^3)$.

Porém a busca local de menor complexidade possui somatórios e cálculos dependentes das posições que serão trocadas e, como cada formiga provavelmente terá dois valores de posições diferentes, as posições de memória acessadas por cada formiga acaba sendo aleatório, o que é ruim para uma implementação em CUDA. Então a busca local de complexidade $O(n^4)$ foi implementada devido a sua previsibilidade de acesso à memória, o que resulta em um melhor desempenho.

A complexidade deste algoritmo é $O(n^3)$ para cada formiga, devido à busca local. Modificar as soluções, atualizar a trilha, escolher soluções e mudar os mecanismos da colônia têm no máximo complexidade $O(n^2)$, mas a complexidade do algoritmo implementado em CUDA é $O(n^4)$.

Se a busca local não for executada, a complexidade do algoritmo passa a ser $O(R * n)$, onde R é o número de trocas que serão feitas na permutação.

O Algoritmo 2 apresenta o pseudocódigo de uma formiga HAS-QAP.

Este algoritmo já foi implementado em um trabalho anterior [4], porém o código era ingênuo: não levou em conta vários pontos de otimização na execução em CUDA. A diferença de tempo entre esta implementação e a atual, assim como comparações e análise dos resultados obtidos são mostradas na seção seguinte.

## 5.3 RESULTADOS EXPERIMENTAIS

Os parâmetros do algoritmo de colônia de formigas para o QAP foram variados para encontrar a melhor configuração. O número de formigas e colônias foram: 32/2, 128/4, 256/8. O parâmetro que indica qual a porcentagem de posições da soluções que sofrerão *swap* foi testado com vários valores, porém o valor padrão 0.3 definido por [103] foi confirmado como melhor. O

**Algoritmo 2** *Algoritmo HAS-QAP*

1: Gere as $m$ soluções iniciais, uma para cada formiga.
2: Inicialize a trilha de feromônios.
3: **para** i de 1 até o número máximo de rodadas **faça**
4: **para** cada formiga $k = 1, \ldots, m$ **faça**
5: Modifique a solução atual associada a k utilizando a trilha de feromônios.
6: Faça uma busca local na solução modificada.
7: Associe uma nova solução à formiga k utilizando um mecanismo de intensificação.
8: **fim-para**
9: Atualiza a tabela utilizando a melhor solução da colônia e a melhor solução gerada nesta rodada pela colônia.
10: Execute o mecanismo de diversificação.
11: **fim-para**

fator de evaporação também foi testado para vários valores, e novamente o fator 0.1 foi o melhor. O mesmo aconteceu com o parâmetro que determina a quantidade de feromônio será depositada na atualização e o parâmetro que indica a porcentagem de *swaps* que utilizarão o método *exploitation*, ambos obtinham melhor resultado quando o valor era igual ao sugerido por [103]: 0.1.

Existe mais um parâmetro importante do algoritmo, a realização de busca local ou não. A busca local é a etapa que consome mais tempo na execução dos algoritmos, como será visto mais a frente, porém traz um grande ganho na qualidade da solução.

As instâncias utilizadas nos testes foram baseadas nas escolhidas por Gambardella, Taillard e Dorigo [103]. Todas as instâncias foram retiradas de [104], assim como suas soluções ótimas ou melhores encontradas até o momento.

As dez primeiras instâncias (de bur26 até tai80b) são instâncias irregulares, que são instâncias onde pelo menos uma das matrizes possuem valores que variam muito, por exemplo, possuem vários valores iguais a zero. Normalmente problemas reais apresentam esta característica. As dez últimas instâncias são de problemas regulares, onde os valores das matrizes são próximos ou uniformemente distribuídos. Meta-heurísticas podem não encontrar boas soluções em problemas irregulares devido à sua característica. As instâncias foram classificadas utilizando uma estatística simples chamada "*flow-dominance*" [103].

O tempo mostrado nas tabelas é medido em milissegundos e foi obtido através da média aritmérica após 10 execuções para cada instância, assim como todos os outros dados. Para medir a distância de uma solução à ótima, a medida de porcentagem de desvio médio (PDM) é utilizada. Este valor pode ser calculado da seguinte maneira: 100∗ (valor da solução obtida − valor da solução ótima) / valor da solução ótima. Quanto menor o PDM,

melhor. Também foram contabilizados em quantas das 10 execuções a solução ótima foi encontrada, este valor é representado pelas colunas OPT.

Tabela 5.1: Resultados das execuções com e sem busca local com 256 formigas em 8 colônias para as instâncias selecionadas.

| | | 100 rodadas com busca local | | | 100 rodadas sem busca local | | |
|---|---|---|---|---|---|---|---|
| Nome da instância | n | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| bur26a | 26 | 2020 | 0.0000 | 10 | 22 | 1.0183 | 0 |
| chr25a | 25 | 1806 | 0.0000 | 10 | 22 | 126.5490 | 0 |
| els19 | 19 | 867 | 0.0000 | 10 | 16 | 9.2916 | 0 |
| kra30a | 30 | 3038 | 0.0000 | 10 | 28 | 19.7514 | 0 |
| tai20b | 20 | 981 | 0.0000 | 10 | 18 | 4.7486 | 0 |
| tai30b | 30 | 3040 | 0.0000 | 10 | 28 | 12.9713 | 0 |
| tai40b | 40 | 7844 | 0.0000 | 10 | 39 | 25.0346 | 0 |
| tai50b | 50 | 18622 | 0.0014 | 0 | 51 | 24.9821 | 0 |
| tai60b | 60 | 32780 | 0.0024 | 0 | 63 | 28.2851 | 0 |
| tai80b | 80 | 88139 | 0.0075 | 0 | 99 | 29.5638 | 0 |
| nug20 | 20 | 984 | 0.0000 | 10 | 18 | 7.3307 | 0 |
| nug30 | 30 | 3038 | 0.0065 | 9 | 28 | 13.0046 | 0 |
| sko42 | 42 | 9482 | 0.0000 | 10 | 40 | 13.6643 | 0 |
| sko72 | 72 | 63049 | 0.0631 | 0 | 85 | 13.0760 | 0 |
| tai20a | 20 | 984 | 0.2737 | 3 | 18 | 8.5497 | 0 |
| tai30a | 30 | 3040 | 0.5671 | 1 | 28 | 10.2363 | 0 |
| tai40a | 40 | 7834 | 1.9371 | 0 | 39 | 12.3202 | 0 |
| tai50a | 50 | 18722 | 3.1779 | 0 | 51 | 12.7912 | 0 |
| tai60a | 60 | 32709 | 3.4494 | 0 | 63 | 12.4221 | 0 |
| tai80a | 80 | 88992 | 3.4116 | 0 | 98 | 11.8598 | 0 |

Na Tabela 5.1 os resultados para as instâncias quando executados por 256 formigas em 8 colônias com e sem busca local são mostrados. Pode-se notar que a busca local tem um grande impacto na execução, tanto na qualidade da solução quanto no tempo de execução. O tempo de execução é muito pequeno - 98 milissegundos no pior caso - mas a qualidade das soluções é muito ruim, 126.5490 para a instância difícil chr25 e 11.8598 para a instância tai80a, que é a instância que consome mais tempo. Isto acontece pois a busca local explora muitas soluções. Enquanto sem a busca local cada formiga gera uma solução e não deriva nenhuma outra desta, com a busca local cada formiga gera uma solução e explora $n^2$soluções derivadas da gerada. Portanto se uma solução muito próxima da ótima deve ser encontrada a busca local deve ser utilizada, mas se a proximidade à solução ótima não é importante, o algoritmo sem busca local encontra muito rapidamente uma solução.

Como consequência de a qualidade das soluções encontradas pelo algoritmo sem busca local serem ruins, estas execuções serão ocultadas das tabelas comparativas seguintes. Nas Figuras 5.3, 5.4 e 5.5 a diferença entre a qualidade com e sem busca local pode ser vista com facilidade.

Tabela 5.2: Resultados da execução para as instâncias selecionadas com 32 formigas em 2 colônias.

| Nome da instância | n | 10 rodadas com busca local | | | 100 rodadas com busca local | | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| bur26a | 26 | 101 | 0.0452 | 3 | 1003 | 0.0338 | 6 |
| chr25a | 25 | 89 | 18.3246 | 0 | 897 | 6.4858 | 1 |
| els19 | 19 | 42 | 0.0000 | 10 | 427 | 0.0000 | 10 |
| kra30a | 30 | 150 | 1.4364 | 3 | 1498 | 1.1001 | 2 |
| tai20b | 20 | 49 | 0.0453 | 0 | 484 | 0.0000 | 10 |
| tai30b | 30 | 149 | 0.1515 | 0 | 1499 | 0.0005 | 0 |
| tai40b | 40 | 323 | 0.1227 | 0 | 3225 | 0.0028 | 0 |
| tai50b | 50 | 615 | 0.3005 | 0 | 6160 | 0.1328 | 0 |
| tai60b | 60 | 1048 | 0.1919 | 0 | 10504 | 0.0417 | 0 |
| tai80b | 80 | 2404 | 0.9357 | 0 | 24041 | 0.3618 | 0 |
| nug20 | 20 | 49 | 0.2957 | 6 | 485 | 0.0000 | 10 |
| nug30 | 30 | 148 | 0.7805 | 0 | 1498 | 0.0980 | 3 |
| sko42 | 42 | 374 | 0.6704 | 0 | 3732 | 0.2036 | 3 |
| sko72 | 72 | 1777 | 0.7601 | 0 | 17808 | 0.3601 | 0 |
| tai20a | 20 | 48 | 1.7969 | 0 | 485 | 0.7653 | 0 |
| tai30a | 30 | 149 | 2.9943 | 0 | 1494 | 1.3700 | 0 |
| tai40a | 40 | 324 | 3.7505 | 0 | 3224 | 3.0175 | 0 |
| tai50a | 50 | 616 | 4.1629 | 0 | 6160 | 3.3942 | 0 |
| tai60a | 60 | 1045 | 4.1325 | 0 | 10489 | 3.7103 | 0 |
| tai80a | 80 | 2408 | 3.9589 | 0 | 24068 | 3.6692 | 0 |

É importante notar que o que define a qualidade das soluções encontradas não é apenas o tamanho do problema, pois, por exemplo, a instância chr25, que tem $n = 25$ foi a que obteve as piores soluções entre todas as instâncias, mesmo sendo relativamente pequena. Isto acontece pois esta instância é extremamente irregular.

Com cem rodadas nota-se que a solução ótima é encontrada diversas vezes, até mesmo nos dez testes realizados e para as duas configurações, como nas instâncias nug20 e tai40b. Na configuração de 256 formigas e 100 rodadas nota-se que a qualidade das soluções encontradas foi extremamente alta, obtendo PDM acima de 0.6 apenas nas instâncias regulares de tamanho maior que 30.

Quando os dados apresentados por este trabalho são comparados os resultados da colônia de formigas HAS originais, apresentados em [103], nota-se que, considerando a qualidade das soluções após 10 rodadas, as obtidas por este trabalho são melhores. Por exemplo, para a instância chr25, o algoritmo original obtinha solução de PDM aproximadamente 15.69 em 10 rodadas com 10 formigas, enquanto nos resultados apresentados a configuração com 32 formigas conseguiu uma solução de PDM 18.32 e a configuração com 256 formigas alcançou 11.38. Quando 100 rodadas são executadas, o traba-

Tabela 5.3: Resultados da execução para as instâncias selecionadas com 128 formigas em 4 colônias.

| | | 10 rodadas com busca local | | | 100 rodadas com busca local | | |
|---|---|---|---|---|---|---|---|
| Nome da instância | n | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| bur26a | 26 | 155 | 0.0034 | 8 | 1551 | 0.0000 | 10 |
| chr25a | 25 | 140 | 12.1760 | 1 | 1385 | 1.8651 | 6 |
| els19 | 19 | 66 | 0.0034 | 9 | 656 | 0.0000 | 10 |
| kra30a | 30 | 234 | 0.7705 | 4 | 2344 | 0.1350 | 9 |
| tai20b | 20 | 74 | 0.0000 | 10 | 745 | 0.0000 | 10 |
| tai30b | 30 | 234 | 0.0299 | 0 | 2347 | 0.0000 | 10 |
| tai40b | 40 | 581 | 0.0147 | 0 | 5675 | 0.0000 | 10 |
| tai50b | 50 | 1284 | 0.0977 | 0 | 12720 | 0.0082 | 0 |
| tai60b | 60 | 2239 | 0.0731 | 0 | 22380 | 0.0025 | 0 |
| tai80b | 80 | 5359 | 0.4214 | 0 | 56300 | 0.0225 | 0 |
| nug20 | 20 | 70 | 0.0467 | 7 | 699 | 0.0000 | 10 |
| nug30 | 30 | 219 | 0.2449 | 0 | 2198 | 0.0065 | 9 |
| sko42 | 42 | 647 | 0.4958 | 0 | 6621 | 0.0190 | 7 |
| sko72 | 72 | 4149 | 0.5273 | 0 | 41160 | 0.1482 | 0 |
| tai20a | 20 | 70 | 1.0058 | 1 | 700 | 0.4305 | 1 |
| tai30a | 30 | 219 | 2.4893 | 0 | 2190 | 0.8398 | 0 |
| tai40a | 40 | 546 | 3.2944 | 0 | 5519 | 2.3199 | 0 |
| tai50a | 50 | 1232 | 3.7858 | 0 | 11830 | 3.2268 | 0 |
| tai60a | 60 | 2139 | 3.8997 | 0 | 21419 | 3.4941 | 0 |
| tai80a | 80 | 5644 | 3.7988 | 0 | 56254 | 3.5651 | 0 |

lho original encontra PDM de 3.0, a configuração com 32 formigas 6.48 e a com 256 formigas 0.0.

A diferença das soluções entre as 10 formigas do trabalho original e as 32 deste trabalho ocorre pela maneira como a implementação em CUDA foi feita: alguns mecanismos de diversificação e intensificação foram alterados para aumentar o paralelismo. O tempo de execução não é comparado pois o algoritmo original foi executado em 1999, e desde então os processadores evoluíram em diversas ordens de magnitude.

Para comparar os resultados deste trabalho com os resultados de trabalhos da literatura ([105], [106], [107], [108], [109]) a configuração de 256 formigas e 100 rodadas foi escolhida por ter qualidade e principalmente tempo consumido mais compatíveis, esta configuração será chamada de ACO a partir deste ponto com a finalidade de facilitar a leitura.

As Tabelas 5.5 e 5.6 mostra os resultados da configuração escolhida e os resultados encontrados na literatura sem nenhum fator de correção. Nas Tabelas 5.7 e 5.8 um fator de correção que leva em conta a frequência do processador foi considerado. Este fator é calculado através da divisão da frequência de clock do processador do trabalho dividido pela frequência do processador deste trabalho (3.3GHz). O tempo mostrado na tabela é a mul-

Tabela 5.4: Resultados da execução para as instâncias selecionadas com 256 formigas em 8 colônias.

| Nome da instância | n | 10 rodadas com busca local | | | 100 rodadas com busca local | | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | OPT | Tempo (ms) | PDM | OPT |
| bur26a | 26 | 202 | 0.0139 | 7 | 2020 | 0.0000 | 10 |
| chr25a | 25 | 181 | 11.3804 | 0 | 1806 | 0.0000 | 10 |
| els19 | 19 | 87 | 0.0000 | 10 | 867 | 0.0000 | 10 |
| kra30a | 30 | 304 | 0.4106 | 6 | 3038 | 0.0000 | 10 |
| tai20b | 20 | 98 | 0.0000 | 0 | 981 | 0.0000 | 10 |
| tai30b | 30 | 305 | 0.0014 | 0 | 3040 | 0.0000 | 10 |
| tai40b | 40 | 775 | 0.0010 | 0 | 7844 | 0.0000 | 10 |
| tai50b | 50 | 1872 | 0.0784 | 0 | 18622 | 0.0014 | 0 |
| tai60b | 60 | 3275 | 0.0617 | 0 | 32780 | 0.0024 | 0 |
| tai80b | 80 | 9419 | 0.3299 | 0 | 88139 | 0.0075 | 0 |
| nug20 | 20 | 98 | 0.0000 | 10 | 984 | 0.0000 | 10 |
| nug30 | 30 | 304 | 0.1829 | 0 | 3038 | 0.0065 | 9 |
| sko42 | 42 | 961 | 0.3225 | 1 | 9482 | 0.0000 | 10 |
| sko72 | 72 | 6407 | 0.4664 | 0 | 63049 | 0.0631 | 0 |
| tai20a | 20 | 99 | 0.9790 | 0 | 984 | 0.2737 | 3 |
| tai30a | 30 | 305 | 2.2071 | 0 | 3040 | 0.5671 | 1 |
| tai40a | 40 | 779 | 3.2582 | 0 | 7834 | 1.9371 | 0 |
| tai50a | 50 | 1877 | 3.6657 | 0 | 18722 | 3.1779 | 0 |
| tai60a | 60 | 3274 | 3.7846 | 0 | 32709 | 3.4494 | 0 |
| tai80a | 80 | 8908 | 3.7156 | 0 | 88992 | 3.4116 | 0 |

tiplicação do fator vezes o tempo original de cada trabalho. Este tempo com fator de correção representa uma aproximação do tempo de execução se o algoritmo fosse executado na mesma máquina que este trabalho, porém este fator não é completamente justo, pois a implementação em CUDA não utiliza o clock do processador principal, e não há como fazer uma estimativa da quantidade de tempo consumida por um código sequencial se fosse executado em uma *GPGPU*, pois isto depende da implementação e das características do algoritmo. Posições das tabelas que contém "-" são posições onde os dados não foram informados no trabalho da literatura.

Os resultados de [109] foram ocultados da tabela sem fator de correção pois apenas quatro instâncias em comum foram executadas (nug20, nug30, sko42 e sko72), porém como os resultados de [108] foram ocultados da tabela com fator de correção pois o trabalho não informou qual a frequência do processador utilizado para os testes, os dados com fator de correção de [109] foram incluídos.

Em comparação com a implementação da meta-heurística *simulated annealing* (SA) de [105], pode-se notar que o tempo consumido pelo SA é sempre maior que ACO e a qualidade das soluções do SA é melhor para as quatro maiores instâncias regulares (de tai40a a tai80a). Para a pequena instância

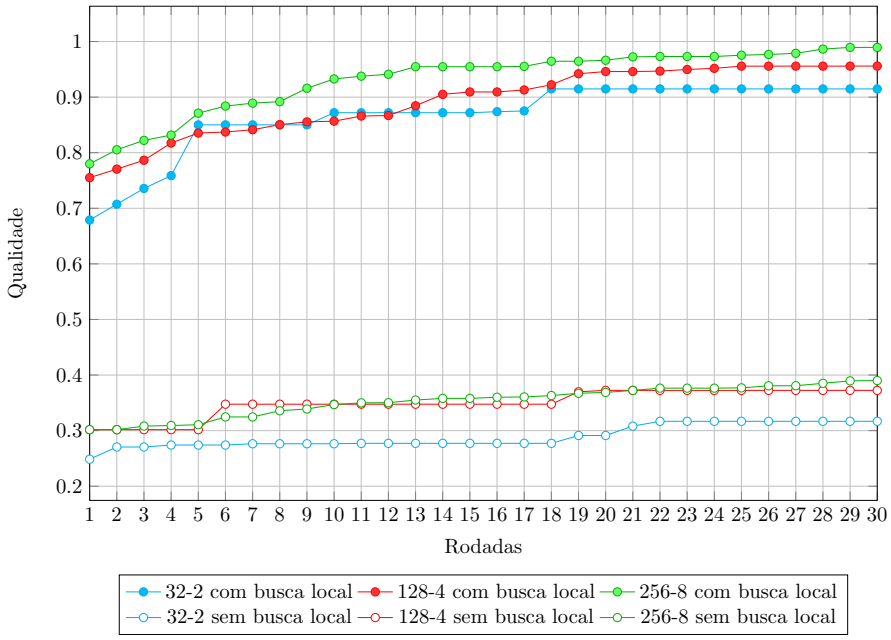



Figura 5.3: Gráfico de aproximação das soluções geradas à ótima em relação ao número de rodadas para a instância chr25 do QAP.

Tabela 5.5: Primeira tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura.

| | | 256-8 | | SA [105] | | ITS [106] | |
|---|---|---|---|---|---|---|---|
| Nome da instância | n | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| bur26a | 26 | 2020 | 0.0000 | - | - | - | - |
| chr25a | 25 | 1806 | 0.0000 | - | - | 900 | 0.18 |
| els19 | 19 | 867 | 0.0000 | - | - | 100 | 0.0 |
| kra30a | 30 | 3038 | 0.0000 | - | - | 1500 | 0.01 |
| tai20b | 20 | 981 | 0.0000 | - | - | 400 | 0.0 |
| tai30b | 30 | 3040 | 0.0000 | - | - | 2500 | 0.01 |
| tai40b | 40 | 7844 | 0.0000 | - | - | 12600 | 0.01 |
| tai50b | 50 | 18622 | 0.0014 | - | - | 33000 | 0.02 |
| tai60b | 60 | 32780 | 0.0024 | - | - | 64000 | 0.04 |
| tai80b | 80 | 88139 | 0.0075 | 1147280 | 0.717 | 180000 | 0.23 |
| nug20 | 20 | 984 | 0.0000 | 6210 | 0.0000 | - | - |
| nug30 | 30 | 3038 | 0.0065 | - | - | 4300 | 0.0 |
| sko42 | 42 | 9482 | 0.0000 | - | - | 9500 | 0.0 |
| sko72 | 72 | 63049 | 0.0631 | 1304260 | 0.188 | 180000 | 0.06 |
| tai20a | 20 | 984 | 0.2737 | - | - | 1800 | 0.0 |
| tai30a | 30 | 3040 | 0.5671 | 72640 | 0.680 | 12000 | 0.0 |
| tai40a | 40 | 7834 | 1.9371 | 233200 | 1.349 | 78000 | 0.22 |
| tai50a | 50 | 18722 | 3.1779 | 462588 | 1.803 | 330000 | 0.41 |
| tai60a | 60 | 32709 | 3.4494 | 948610 | 1.930 | 750000 | 0.45 |
| tai80a | 80 | 88992 | 3.4116 | 1242470 | 1.487 | 3600000 | 0.36 |

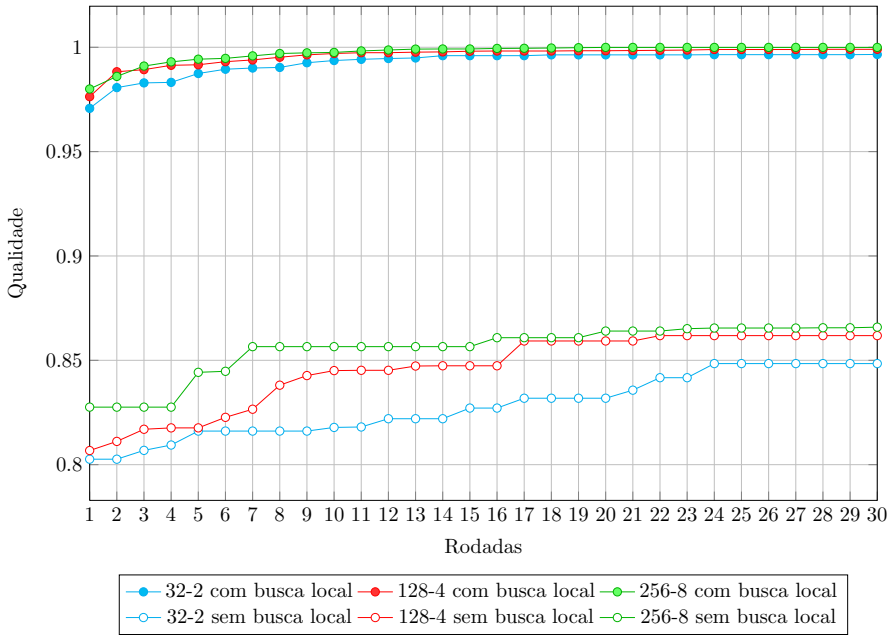



Figura 5.4: Gráfico de aproximação das soluções geradas à ótima em relação ao número de rodadas para a instância nug30 do QAP.

Tabela 5.6: Segunda tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura.

| | | 256-8 | | TS [107] | | SC-TS [108] | |
|---|---|---|---|---|---|---|---|
| Nome da instância | n | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| bur26a | 26 | 2020 | 0.0000 | - | - | 540 | 0.01 |
| chr25a | 25 | 1806 | 0.0000 | - | - | - | - |
| els19 | 19 | 867 | 0.0000 | - | - | 80 | 3.822 |
| kra30a | 30 | 3038 | 0.0000 | - | - | 2880 | 0.714 |
| tai20b | 20 | 981 | 0.0000 | - | - | 150 | 0.335 |
| tai30b | 30 | 3040 | 0.0000 | 250 | 1.62 | 2080 | 0.313 |
| tai40b | 40 | 7844 | 0.0000 | 290 | 3.07 | 5520 | 0.219 |
| tai50b | 50 | 18622 | 0.0014 | 400 | 5.15 | 14070 | 0.281 |
| tai60b | 60 | 32780 | 0.0024 | - | - | 24880 | 0.886 |
| tai80b | 80 | 88139 | 0.0075 | 820 | 3.91 | 60240 | 0.798 |
| nug20 | 20 | 984 | 0.0000 | 160 | 1.56 | - | - |
| nug30 | 30 | 3038 | 0.0065 | 220 | 2.65 | 2880 | 0.022 |
| sko42 | 42 | 9482 | 0.0000 | - | - | 8300 | 0.016 |
| sko72 | 72 | 63049 | 0.0631 | - | - | 43730 | 0.159 |
| tai20a | 20 | 984 | 0.2737 | - | - | 530 | 0.246 |
| tai30a | 30 | 3040 | 0.5671 | 250 | 1.62 | 4480 | 0.154 |
| tai40a | 40 | 7834 | 1.9371 | 320 | 6.12 | 11980 | 0.561 |
| tai50a | 50 | 18722 | 3.1779 | 390 | 6.49 | 14050 | 0.889 |
| tai60a | 60 | 32709 | 3.4494 | - | - | 24670 | 0.940 |
| tai80a | 80 | 88992 | 3.4116 | 790 | 6.66 | 60340 | 0.648 |

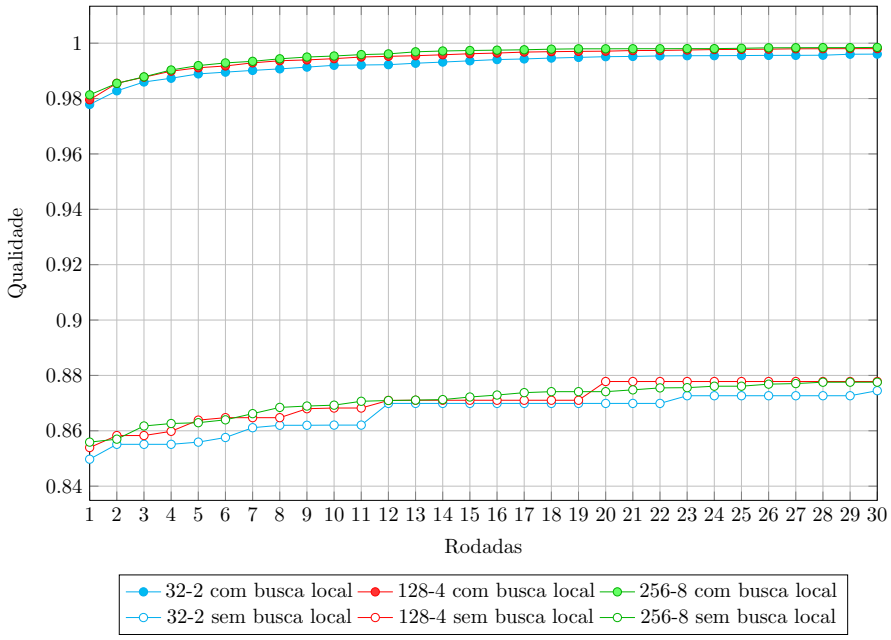



Figura 5.5: Gráfico de aproximação das soluções geradas à ótima em relação ao número de rodadas para a instância sko72 do QAP.

Tabela 5.7: Primeira tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura com fator de correção no tempo de execução.

| Nome da instância | n | 256-8 | | SA [105] | | ITS [106] | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| bur26a | 26 | 2020 | 0.0000 | - | - | - | - |
| chr25a | 25 | 1806 | 0.0000 | - | - | 245 | 0.18 |
| els19 | 19 | 867 | 0.0000 | - | - | 27 | 0.0 |
| kra30a | 30 | 3038 | 0.0000 | - | - | 409 | 0.01 |
| tai20b | 20 | 981 | 0.0000 | - | - | 109 | 0.0 |
| tai30b | 30 | 3040 | 0.0000 | - | - | 681 | 0.01 |
| tai40b | 40 | 7844 | 0.0000 | - | - | 3436 | 0.01 |
| tai50b | 50 | 18622 | 0.0014 | - | - | 8999 | 0.02 |
| tai60b | 60 | 32780 | 0.0024 | - | - | 17452 | 0.04 |
| tai80b | 80 | 88139 | 0.0075 | 924707 | 0.717 | 49086 | 0.23 |
| nug20 | 20 | 984 | 0.0000 | 5005 | 0.0000 | - | - |
| nug30 | 30 | 3038 | 0.0065 | - | - | 1172 | 0.0 |
| sko42 | 42 | 9482 | 0.0000 | - | - | 2590 | 0.0 |
| sko72 | 72 | 63049 | 0.0631 | 1051233 | 0.188 | 49086 | 0.06 |
| tai20a | 20 | 984 | 0.2737 | - | - | 490 | 0.0 |
| tai30a | 30 | 3040 | 0.5671 | 58547 | 0.680 | 3272 | 0.0 |
| tai40a | 40 | 7834 | 1.9371 | 187959 | 1.349 | 21270 | 0.22 |
| tai50a | 50 | 18722 | 3.1779 | 372845 | 1.803 | 89991 | 0.41 |
| tai60a | 60 | 32709 | 3.4494 | 764579 | 1.930 | 204525 | 0.45 |
| tai80a | 80 | 88992 | 3.4116 | 1001430 | 1.487 | 981720 | 0.36 |

Tabela 5.8: Segunda tabela comparativa de PDM e tempo de execução entre a configuração de 256 formigas em 8 colônias com busca local e dados encontrados na literatura com fator de correção no tempo de execução.

| Nome da instância | n | 256-8 | | TS [107] | | SL [109] | |
|---|---|---|---|---|---|---|---|
| | | Tempo (ms) | PDM | Tempo (ms) | PDM | Tempo (ms) | PDM |
| bur26a | 26 | 2020 | 0.0000 | - | - | - | - |
| chr25a | 25 | 1806 | 0.0000 | - | - | - | - |
| els19 | 19 | 867 | 0.0000 | - | - | - | - |
| kra30a | 30 | 3038 | 0.0000 | - | - | - | - |
| tai20b | 20 | 981 | 0.0000 | - | - | - | - |
| tai30b | 30 | 3040 | 0.0000 | 191 | 1.62 | - | - |
| tai40b | 40 | 7844 | 0.0000 | 222 | 3.07 | - | - |
| tai50b | 50 | 18622 | 0.0014 | 306 | 5.15 | - | - |
| tai60b | 60 | 32780 | 0.0024 | - | - | - | - |
| tai80b | 80 | 88139 | 0.0075 | 628 | 3.91 | - | - |
| nug20 | 20 | 984 | 0.0000 | 122 | 1.56 | 15513 | 0.013 |
| nug30 | 30 | 3038 | 0.0065 | 168 | 2.65 | 65932 | 0.0006 |
| sko42 | 42 | 9482 | 0.0000 | - | - | 170951 | 0.42 |
| sko72 | 72 | 63049 | 0.0631 | - | - | 2507385 | 0.26 |
| tai20a | 20 | 984 | 0.2737 | - | - | - | - |
| tai30a | 30 | 3040 | 0.5671 | 191 | 1.62 | - | - |
| tai40a | 40 | 7834 | 1.9371 | 245 | 6.12 | - | - |
| tai50a | 50 | 18722 | 3.1779 | 298 | 6.49 | - | - |
| tai60a | 60 | 32709 | 3.4494 | - | - | - | - |
| tai80a | 80 | 88992 | 3.4116 | 605 | 6.66 | - | - |

nug20, ambas implementações encontram soluções ótimas frequentemente, porém o tempo consumido pelo SA é aproximadamente cinco vezes maior. Para a instância tai80a a qualidade encontrada pelo SA é muito maior do que a encontrada pelo ACO (1.49 contra 3.41), porém o tempo consumido é mais de dez vezes maior (890 ms contra 1.001.430 ms). Para a instância tai80b, a solução encontrada pelo ACO é melhor (0.0075 contra 0.717) e o tempo consumido é mais de dez vezes menor (88.140 contra 924.707).

Quando os resultados da implementação de uma busca tabu iterada (ITS) de [106] são comparados, nota-se que, com o fator de correção, o tempo consumido pelo ITS é sempre menor para as instâncias irregulares, porém a qualidade das soluções é um pouco menor. Já para as instâncias regulares a qualidade das soluções obtidas pelo ITS são melhores em todos os casos (apenas para a instância sko72 elas são iguais) e o tempo consumido pelo ITS é menor para as instâncias de nug20 a tai20a, e maior para as outras instâncias. O fator de correção causou um grande impacto nestes resultados pois o processador utilizado em [106] tinha frequência de 900 MHz apenas. Este algoritmo deve ser estudado pois a qualidade das soluções encontradas é extremamente alta, tanto é que o trabalho que o propôs encontrou soluções melhores do que as previamente conhecidas para algumas instâncias

do QAPLIB [104], portanto algumas instâncias tiveram seu valor de melhor solução encontrada atualizados.

A implementação da meta-heurística busca tabu (TS) implementada em [107], considerando a tabela com fator de correção, consumiu menos tempo em todas as instâncias do que todas as implementações comparadas, porém a qualidade das soluções foi extremamente menor, por exemplo, para a instância nug30 enquanto o ACO obteve PDM 0.006 e o ITS 0.0, o TS obteve PDM igual a 2.65. O mesmo acontece para a instância tai40b, o ACO obteve 0.0 e o ITS 0.01, já o TS obteve 3.07.

Ao considerar os resultados de [108], que implementou uma meta-heurística de busca tabu auto-controlada (SC-TS), sem o fator de correção, nota-se que a qualidade das soluções encontradas pelo ACO são melhores para todas as instâncias irregulares e algumas regulares (nug30, sko42 e sko72). Ao comparar o tempo de execução, para a maioria das instâncias o SC-TS consumiu menos tempo, com exceção das instâncias tai40a e tai30a.

Em [109] um algoritmo de suavização Lagrangeana (SL) foi implementado. Apenas quatro instâncias em comum com este trabalho foram executadas: nug20, nug30, sko42 e sko72. Levando em consideração os dados com o fator de correção, nota-se que o tempo consumido pelo SL foi muito maior para todas as instâncias e a qualidade foi melhor apenas para a instância nug30 (0.0006 contra 0.0065) que consumiu aproximadamente 21 vezes mais tempo que o ACO.

Ao comparar os resultados da implementação deste trabalho com a implementação anterior [4], nota-se que o tempo melhorou bruscamente. Os *speedups* são mostrados na Tabela 5.9. Na implementação anterior, cada formiga era representada por uma *thread*, portanto com a configuração de 32 formigas e 2 colônias, em uma rodada apenas um *kernel* com 32 *threads* eram lançadas e executavam o algoritmo de cada formiga. Na implementação atual, em uma rodada vários *kernels* são lançados, com diferentes configurações de *grid*. Por exemplo, para fazer uma troca de posição na solução, um *kernel* com $m$ blocos, onde $m$ é o número de formigas, cada bloco com $n$ *threads* cada é lançado, fazendo com que a granularidade do algoritmo seja muito menor. A qualidade da solução não é comparada, pois o algoritmo é exatamente o mesmo, apenas mudando a implementação.

Nota-se que o número de formigas é determinante na qualidade das soluções encontradas. Na grande maioria das instâncias quanto maior o número de formigas, melhor a solução a cada rodada. Pequenas irregularidades podem ser vistas nas Figuras 5.3 e 5.5 devido à natureza estocástica do algoritmo.

Tabela 5.9: *Speedup* dos resultados da implementação atual em relação à implementação anterior [4], ambos com busca local.

| Nome da instância | n | 10 rodadas | | | 100 rodadas | | |
|---|---|---|---|---|---|---|---|
| | | 32-2 | 128-4 | 256-8 | 32-2 | 128-4 | 256-8 |
| bur26a | 26 | 38.6360 | 25.8062 | 21.1802 | 38.3840 | 25.6126 | 21.0453 |
| chr25a | 25 | 38.3120 | 25.2901 | 20.8637 | 38.1287 | 25.3914 | 20.9503 |
| els19 | 19 | 35.7864 | 23.8557 | 19.5042 | 35.4790 | 23.7468 | 19.4363 |
| kra30a | 30 | 39.4460 | 25.6153 | 21.4672 | 39.2460 | 25.8530 | 21.3596 |
| tai20b | 20 | 36.4301 | 24.5133 | 19.9150 | 36.2023 | 24.3852 | 19.8768 |
| tai30b | 30 | 39.4609 | 26.0308 | 21.4490 | 39.2258 | 25.8378 | 21.3131 |
| tai40b | 40 | 44.0672 | 24.5663 | 19.8839 | 43.7137 | 25.0262 | 19.8459 |
| tai50b | 50 | 46.1901 | 22.5414 | 16.7932 | 45.7606 | 22.4206 | 16.7234 |
| tai60b | 60 | 47.0006 | 22.3557 | 16.5248 | 46.5559 | 22.1746 | 16.3647 |
| tai80b | 80 | 48.7396 | 20.7666 | 14.3327 | 48.3776 | 19.4939 | 13.7616 |
| nug20 | 20 | 36.5219 | 24.4874 | 19.9933 | 36.4383 | 24.4137 | 19.9548 |
| nug30 | 30 | 39.4735 | 26.0647 | 21.6182 | 39.3102 | 25.9050 | 21.5387 |
| sko42 | 42 | 44.6852 | 25.1173 | 19.2135 | 44.4164 | 25.5390 | 19.2045 |
| sko72 | 72 | 47.6826 | 20.4263 | 14.2500 | 47.4278 | 19.8097 | 14.2120 |
| tai20a | 20 | 36.3487 | 24.3961 | 20.0023 | 36.3527 | 24.3873 | 19.9649 |
| tai30a | 30 | 39.4262 | 25.8908 | 21.4860 | 39.1996 | 25.8475 | 21.4060 |
| tai40a | 40 | 44.0325 | 24.7238 | 19.9751 | 43.7556 | 24.3238 | 19.7583 |
| tai50a | 50 | 46.2554 | 22.8187 | 16.7834 | 45.8607 | 22.3275 | 16.6374 |
| tai60a | 60 | 46.9627 | 22.5030 | 16.6633 | 45.5064 | 21.9579 | 16.4962 |
| tai80a | 80 | 48.5940 | 20.1448 | 14.0835 | 48.0903 | 20.4981 | 14.1400 |

# 6 CONCLUSÃO E TRABALHOS FUTUROS

Se soluções próximas da ótima são suficientes para uma aplicação uma meta-heurística como os sistemas de colônias de formigas pode ser utilizada, pois além de encontrarem soluções boas, não consomem muito tempo. Ao utilizar um algoritmo como os apresentados neste trabalho, deve-se equilibrar o número de formigas, número de rodadas e utilização ou não de busca local, dependendo da qualidade da solução desejada, que é o mesmo que equilibrar tempo e qualidade da solução. Quanto melhor a solução desejada, mais tempo será consumido. Se não existe um algoritmo exato para o problema mas é desejado uma solução muito próxima da ótima, podemos configurar o algoritmo para trabalhar com um alto número de formigas e colônias, fazer com que realizem uma busca local, se possível, em várias rodadas. Esta configuração irá obter soluções muito próximas da ótima, porém consumirá uma quantidade de tempo grande, como visto nos testes experimentais. Já se a solução desejada deve ser calculada muito rapidamente e a qualidade desta não é o fator mais importante, podemos configurar poucas formigas, sem busca local, com poucas rodadas.

Para o algoritmo de otimização de formigas para o MKP, a implementação do método de programação dinâmica híbrida com uma melhoria no cálculo de limitantes inferiores(HDP+LBC) de [7] obteve melhores resultados (qualidade de soluções maior e tempo menor) que este trabalho para quase todas as instâncias do segundo conjunto da biblioteca OR. Quando os outros trabalhos são comparados os resultados não são tão ruins. O *kernel search* de [91] obtêm soluções com qualidade altíssima, muito próxima da ótima, porém o tempo consumido é até 575 vezes maior que o tempo deste trabalho. A implementação de um método exato (*branch-and-bound*) de [92] mostra o quão difícil é encontrar uma solução ótima, pois mesmo para uma instância não muito grande ($250 \times 10$) pode consumir até 10 horas. Finalmente, em [93], várias meta-heurísticas foram comparadas e uma heurística de redução de problemas sugerida. Como os tempos não são mostrados não se pode concluir nada sobre estes algoritmos.

Em relação ao QAP, o algoritmo de otimização de formigas implementado obteve resultados melhores para todas menos quatro tamanhos de instâncias em comparação ao algoritmo *simulated annealing* de [105]. Nas instâncias onde o SA obteve melhores soluções o tempo consumido foi até dez vezes maior. Já a busca tabu iterada de [106] obteve soluções de qualidade um

pouco piores nas instâncias irregulares e soluções e tempos melhores para as regulares. A busca tabu de [107] encontrou soluções de qualidade ruim quando comparadas a este e outros trabalhos, porém o tempo consumido foi muito pequeno. Este trabalho obteve melhores resultados do que a busca tabu auto-controlada de [108] na maioria das instâncias e a quantidade de instâncias em comum executadas por [109] não é suficiente para alguma conclusão.

Os resultados obtidos por este trabalho para o MKP foram piores do que as obtidas por outros trabalhos da literatura principalmente pelo fato de que não há um processo de busca local como há para o QAP. Tal busca local é difícil de ser implementada para o MKP pois deve-se modificar uma solução e explorar o espaço de soluções próximo, porém, ao modificar uma solução do MKP, pode-se infringir uma das equações do problema. Por exemplo, ao remover um objeto de uma solução do MKP, pode ser que nenhum outro objeto caiba no espaço liberado ou pode ser que vários outros objetos caibam neste espaço liberado. Portanto esta possível busca local para o MKP não é um processo determinístico, para cada formiga um número diferente de passos de remoção e adição de objetos deve ser feito e isto causa problemas de desempenho em códigos paralelos para GPGPU.

A existência de bibliotecas como o QAPLIB [104] e a biblioteca OR [84] ajuda na expansão do estudo sobre estes problemas, já que as instâncias podem ser utilizadas para a comparação de algoritmos e implementações. O problema da mochila binária não possui uma biblioteca deste tipo, portanto comparações são difíceis de serem feitas. Por exemplo, o algoritmo de programação dinâmica em CUDA foi baseado no algoritmo proposto em [94]

Ao comparar os resultados do algoritmo de otimização de formigas para o QAP, pode-se claramente ver que apenas implementar um algoritmo em paralelo, no caso deste trabalho utilizando CUDA, não garante uma melhora no tempo de execução. A implementação deve ser realizada cuidadosamente e levar em conta detalhes do *hardware*, como acesso à memória eficiente, com coalescência, alta vazão de dados, lançar um número suficiente de *threads* para ocupar o máximo de multiprocessadores possíveis e diminuir a granularidade do algoritmo. A diminuição da granularidade, além de melhorar a organização do código e o desempenho, também faz com que um problema que pode ocorrer na execução do código no sistema operacional Windows seja solucionado. Se a execução de um *kernel* ultrapassar um certo limite (por padrão são dois segundos), ou seja, se a placa de vídeo não devolver a prioridade de execução para o sistema operacional, o *kernel* é abortado pelo SO.

Como trabalhos futuros, algumas extensões dos algoritmos implementados devem ser feitas. No caso do QAP, o algoritmo deve ser modificado para resolver também o problema quadrático de alocação generalizado [110] e outras variações do QAP. No caso do problema da mochila, diversos problemas já citados na seção correspondente podem ter o algoritmo modificado para solucioná-los. Ainda, outros métodos que aparentam obter ganhos quando implementados em paralelo, principalmente em *GPGPUs*, podem ser imple-

mentados e comparados ao algoritmo de otimização de formigas. Como exemplo destes métodos temos os algoritmos *multi-start* [111], algoritmos genéticos [35, 36], otimização de enxame de partículas [44] entre outros.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] R. E. Burkard, E. Çela, P. M. Pardalos, and L. Pitsoulis, "The quadratic assignment problem," in *Handbook of Combinatorial Optimization*, P. P. Pardalos and M. G. C. Resende, Eds. Dordrecht: Kluwer Academic Publishers, 1998, pp. 241–238.

[2] E. Talbi, *Metaheuristics: From Design to Implementation*, ser. Wiley Series on Parallel and Distributed Computing. Wiley, 2009, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=SIsa6zi5XV8C

[3] R. Doe. (2006, may) File:aco branches.svg. Último acesso em 23/03/2013. [Online]. Disponível em: http://commons.wikimedia.org/wiki/File:Aco_branches.svg

[4] E. Cáceres, H. Fingler, H. Mongelli, and S. Song, "Ant colony system based solutions to the quadratic assignment problem on gpgpu," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, sept. 2012, pp. 314 –322.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[6] D. Pisinger, "Algorithms for knapsack problems," Ph.D. dissertation, University of Copenhagen, Dept. of Computer Science, 1995.

[7] V. Boyer, M. Elkihel, and D. El Baz, "Heuristics for the 0-1 multidimensional knapsack problem," *European Journal of Operational Research*, vol. 199, no. 3, pp. 658–664, December 2009, Último acesso em 23/03/2013. [Online]. Disponível em: http://ideas.repec.org/a/eee/ejores/v199y2009i3p658-664.html

[8] B. Dietrich and L. Escudero, "Coefficient reduction for knapsack-like constraints in oñ1 programs with variable upper bounds," *Operations Research Letters*, vol. 9, no. 1, pp. 9 – 14, 1990, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/0167637790900343

[9] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer, 2004, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=u5DB7gcko8YC

[10] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[11] M. G. Lagoudakis, "The 0-1 knapsack problem – an introductory survey," Tech. Rep., 1996.

[12] A. N. Elshafei, "Hospital layout as a quadratic assignment problem," *Operations Research Quarterly*, vol. 28, pp. 167–179, 1977.

[13] L. Steinberg, "The backboard wiring problem: a placement algorithm," *SIAM Review*, vol. 3, pp. 37–50, 1961.

[14] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Politecnico di Milano, Italy, 1992.

[15] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldûn, "Enhancing data parallelism for ant colony optimization on gpus," *Journal of Parallel and Distributed Computing*, no. 0, pp. –, 2012, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0743731512000032

[16] A. Turing, "On Computable Numbers with an Application to the Entscheidungs Problem," *Proc. London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[17] R. G. Downey, M. R. Fellows, R. Niedermeier, P. Rossmanith, R. G. D. (wellington, N. Zeal, M. R. F. (newcastle, R. N. (tubingen, and P. R. (tu Munchen, "Parameterized complexity," 1998.

[18] G. Pólya, *How to solve it; a new aspect of mathematical method*. Princeton University Press, 1945, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.worldcat.org/oclc/1628881

[19] V. Kann, "On the Approximability of NP-complete Optimization Problems," Ph.D. dissertation, Royal Institute of Technology Stockholm, 1992.

[20] J. Håstad, "Some optimal inapproximability results," *J. ACM*, vol. 48, no. 4, pp. 798–859, Jul. 2001, Último acesso em 23/03/2013. [Online]. Disponível em: http://doi.acm.org/10.1145/502090.502098

[21] H. Bersini and F. Varela, "Hints for adaptive problem solving gleaned from immune networks," in *Parallel Problem Solving from Nature*, ser. Lecture Notes in Computer Science, H.-P. Schwefel and R. M‰nner, Eds. Springer Berlin Heidelberg, 1991, vol. 496, pp. 343–354, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1007/BFb0029775

[22] J. D. Farmer, N. H. Packard, and A. S. Perelson, "The immune system, adaptation, and machine learning," *Phys. D*, vol. 2, no. 1-3, pp. 187–204, Oct. 1986, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0167-2789(81)90072-5

[23] T. Seeley, *The Wisdom of the Hive: The Social Physiology of Honey Bee Colonies*. Harvard University Press, 1995, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=zhzNJjI2MqAC

[24] Y. Yonezawa and T. Kikuchi, "Ecological algorithm for optimal ordering used by collective honey bee behavior," in *Micro Machine and Human Science, 1996., Proceedings of the Seventh International Symposium*, oct 1996, pp. 249 –256.

[25] S. Prestwich, "Combining the scalability of local search with the pruning techniques of systematic search," *Annals of Operations Research*, vol. 115, pp. 51–72.

[26] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Phys. D*, vol. 42, no. 1-3, pp. 228–234, Jun. 1990, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0167-2789(90)90076-2

[27] P. Husbands and F. Mill, "Simulated co-evolution as the mechanism for emergent planning and scheduling," in *4th International Conference on Genetic Algorithms*, 1991, pp. 264–270, http://www.odysci.com/article/1010112985726497.

[28] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, may 1996, pp. 312 –317.

[29] K. V. Price, "Genetic annealing," vol. 19, no. 11, pp. 127–132, Oct. 1994.

[30] R. Storn and K. Price, "Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces," 1995.

[31] S. Baluja, "Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning," Tech. Rep., 1994.

[32] L. J. Fogel, "Toward inductive inference automata," in *IFIP Congress'62*, 1962, pp. 395–400.

[33] I. Rechenberg, "Cybernetic solution path of an experimental problem," Royal Air Force Establishment, Tech. Rep., 1965.

[34] H.-P. Schwefel, "Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik," Diplomarbeit, Technische Universität Berlin, Hermann Föttinger–Institut für Strömungstechnik, März 1965.

[35] J. H. Holland, "Outline for a logical theory of adaptive systems," *J. ACM*, vol. 9, no. 3, pp. 297–314, Jul. 1962, Último acesso em 23/03/2013. [Online]. Disponível em: http://doi.acm.org/10.1145/321127.321128

[36] ——, *Adaptation in natural and artificial systems.* Cambridge, MA, USA: MIT Press, 1992, Último acesso em 23/03/2013. [Online]. Disponível em: http://portal.acm.org/citation.cfm?id=129194

[37] G. Dueck, "New optimization heuristics: The great deluge algorithm and the record-to-record travel," *Journal of Computational Physics*, vol. 104, no. 1, pp. 86 – 92, 1993, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0021999183710107

[38] C. Voudouris, "Guided local search - an illustrative example in function optimisation," in *In BT Technology Journal, Vol.16, No.3*, 1998, pp. 46–50.

[39] C. Voudouris and E. Tsang, "Guided local search," European Journal of Operational Research, Tech. Rep., 1995.

[40] J. Koza, *Genetic Programming: vol. 1 , On the programming of computers by means of natural selection*, ser. Complex Adaptive Systems Series. Bradford, 1992, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=Bhtx060BV0EC

[41] T. A. Feo and M. G. C. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Oper. Res. Lett.*, vol. 8, no. 2, pp. 67–71, Apr. 1989, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0167-6377(89)90002-3

[42] O. Martin, S. W. Otto, and E. W. Felten, "Large-step markov chains for the traveling salesman problem," *Complex Systems*, vol. 5, pp. 299–326, 1991.

[43] I. Charon and O. Hudry, "The noising method: a new method for combinatorial optimization," *Operations Research Letters*, vol. 14, no. 3, pp. 133 – 137, 1993, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/016763779390023A

[44] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, nov/dec 1995, pp. 1942 –1948 vol.4.

[45] V. Černý, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization*

*Theory and Applications*, vol. 45, no. 1, pp. 41–51, Jan. 1985, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1007/bf00940812

[46] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[47] F. Glover and C. McMillan, "The general employee scheduling problem: an integration of ms and ai," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 563–573, May 1986, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0305-0548(86)90050-X

[48] F. Glover, "Heuristics for integer programming using surrogate constraints," *Decision Sciences*, vol. 8, no. 1, pp. 156–166, 1977, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1111/j.1540-5915.1977.tb01074.x

[49] G. Dueck and T. Scheuer, "Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing," *J. Comput. Phys.*, vol. 90, no. 1, pp. 161–175, Aug. 1990, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0021-9991(90)90201-B

[50] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, May 1986, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/0305-0548(86)90048-1

[51] P. Hansen, "The Steepest Ascent Mildest Descent Heuristic for Combinatorial Programming," in *Proceedings of the Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[52] N. Mladenović, "A variable neighborhood algorithm – a new metaheuristics for combinatorial optimization," in *Abstracts of Papers Presented at Optimization Days. Montr$$al*, 1995.

[53] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics Part B*, vol. 26, pp. 29–41, 1996.

[54] M. Dorigo and L. M. Gambardella, "A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 53–56, 1997.

[55] T. Stutzle and M. Dorigo, "Aco algorithms for the traveling salesman problem," 1999.

[56] T. Stützle, "Max-min ant system for quadratic assignment problems," 1997.

[57] D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens, "Classification with ant colony optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 5, pp. 651 –665, oct. 2007.

[58] V. Maniezzo and M. Milandri, "An ant-based framework for very strongly constrained problems," in *Proceedings of the Third International Workshop on Ant Algorithms*, ser. ANTS '02. London, UK, UK: Springer-Verlag, 2002, pp. 222–227, Último acesso em 23/03/2013. [Online]. Disponível em: http://dl.acm.org/citation.cfm?id=646686.702812

[59] A. V. Donati, R. Montemanni, N. Casagrande, A. E. Rizzoli, and L. M. Gambardella, "Time dependent vehicle routing problem with a multi ant colony system," *European Journal of Operational Research*, vol. 185, no. 3, pp. 1174 – 1191, 2008, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0377221706006345

[60] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[61] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[62] R. Fernando, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[63] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[64] Geforce gtx 570 | specifications | geforce. Último acesso em 23/03/2013. [Online]. Disponível em: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-570/specifications

[65] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[66] J. H. Lorie and L. J. Savage, "Three problems in rationing capital," *The Journal of Business*, vol. 28, p. 229, 1955, Último acesso em 23/03/2013. [Online]. Disponível em: http://ideas.repec.org/a/ucp/jnlbus/v28y1955p229.html

[67] H. M. Markowitz and A. S. Manne, "On the solution of discrete programming problems," *Econometrica*, vol. 25, no. 1, pp. 84+, 1957.

[68] A. Fréville, "The multidimensional 0-1 knapsack problem: An overview," *European Journal of Operational Research*, vol. 155, no. 1, pp. 1 – 21, 2004, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0377221703002741

[69] C. McMillan and D. R. Plane, "Education," *Decision Sciences*, vol. 4, no. 1, pp. 119–132, 1973.

[70] C. C. Petersen, "Computational experience with variants of the balas algorithm applied to the selection of rd projects," *Management Science*, vol. 13, no. 9, pp. 736–750, 1967.

[71] H. Weingartner, *Mathematical Programming and the Analysis of Capital Budgeting Problems: With 3 Related Articles*, ser. Ford Foundation prize dissertation. Markham, 1967, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=igYwuAAACAAJ

[72] P. Gilmore and R. Gomory, "The theory and computation of knapsack functions," *Operations Research*, vol. 15, pp. 1045–1075, 1967.

[73] R. E. Bellman, *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[74] W. Shih, "A branch and bound method for the multiconstraint zero one knapsack problem," *J. Operational Research Society*, vol. 30, no. 4, pp. 369–378, 1979.

[75] C. E. Gearing, W. W. Swart, and T. Var, "Determining the optimal investment policy for the tourism sector of a developing country," *Management Science*, vol. 20, no. 4-Part-I, pp. 487–497, 1973.

[76] B. Gavish and H. Pirkul, "Computer and database location in distributed computer systems," *IEEE Trans. Comput.*, vol. 35, no. 7, pp. 583–590, Jul. 1986.

[77] E. Chajakis and M. Guignard, "Scheduling deliveries in vehicles with multiple compartments," *Journal of Global Optimization*, vol. 26, pp. 43–78, 2003.

[78] A. Straszak, M. Libura, J. Sikorski, and D. Wagner, "Computer-assisted constrained approval voting," *Group Decision and Negotiation*, vol. 2, pp. 375–385, 1993, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1007/BF01384490

[79] M. Vasquez and J.-K. Hao, "A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite," *Comput. Optim. Appl.*, vol. 20, no. 2, pp. 137–157, Nov. 2001.

[80] N. CPLEX Optimization, Inc. Incline Village, *Using the CPLEX Callable Library: Including Using the CPLEX Base System with CPLEX Barrier and Mixed Integer Solver Options; [version 4.0 of "using the CPLEX Callable Library"].* CPLEX Optimization, 1995, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=TdeyZwEACAAJ

[81] I. B. M. Corporation, *IBM Optimization Subroutine Library: Guide and Reference.* IBM Corporation, 1990, Último acesso em 23/03/2013. [Online]. Disponível em: http://books.google.com.br/books?id=IMPfQwAACAAJ

[82] R. Laundy, M. Perregaard, G. Tavares, H. Tipi, and A. Vazacopoulos, "Solving hard mixed-integer programming problems with xpress-mp: A miplib 2003 case study," *INFORMS J. on Computing*, vol. 21, no. 2, pp. 304–313, Apr. 2009, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1287/ijoc.1080.0293

[83] C. Solnon and D. Bridge, "An ant colony optimization meta-heuristic for subset selection problems," in *Systems Engineering using Swarm Particle Optimization*, N. Nedjah and L. M. Mourelle, Eds. Nova Science Publishers, 2006, pp. 7–29.

[84] J. E. Beasley, "OR-Library: distributing test problems by electronic mail," *Journal of the Operational Research Society*, vol. 41, no. 11, pp. 1069–1072, 1990.

[85] A. Freville and G. Plateau, "Hard 0-1 multiknapsack test problems for size reduction methods," *Investigation Operativa*, vol. 1, pp. 251–270, 1990.

[86] M. A. Osorio, F. Glover, and P. Hammer, "Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions," *Annals of Operations Research*, vol. 117, pp. 71–93, 2002.

[87] B. Gavish and H. Pirkul, "Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality," *Mathematical Programming*, vol. 31, pp. 78–105, 1985.

[88] P. Chu and J. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, pp. 63–86, 1998.

[89] A. Fréville and G. Plateau, "An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem," *Discrete Applied Mathematics*, vol. 49, no. 1,Äì3, pp. 189 – 212, 1994, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/0166218X94902097

[90] Mkp Instances - John Drake. Último acesso em 23/03/2013. [Online]. Disponível em: http://www.cs.nott.ac.uk/~jqd/mkp/index.html

[91] E. Angelelli, R. Mansini, and M. G. Speranza, "Kernel search: A general heuristic for the multi-dimensional knapsack problem," *Computers  Operations Research*, vol. 37, no. 11, pp. 2017 – 2026, 2010, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0305054810000328

[92] S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi, and P. Michelon, "A multi-level search strategy for the 0-1 multidimensional knapsack problem," *Discrete Applied Mathematics*, vol. 158, no. 2, pp. 97 – 109, 2010, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0166218X09003345

[93] R. R. Hill, Y. Kun Cho, and J. T. Moore, "Problem reduction heuristic for the 0-1 multidimensional knapsack problem," *Comput. Oper. Res.*, vol. 39, no. 1, pp. 19–26, Jan. 2012, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/j.cor.2010.06.009

[94] V. Boyer, D. El Baz, and M. Elkihel, "Dense dynamic programming on multi gpu," in *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '11.  Washington, DC, USA: IEEE Computer Society, 2011, pp. 545–551, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1109/PDP.2011.25

[95] D. Pisinger, "Core problems in knapsack algorithms." *Operations Research*, vol. 47, pp. 570–575, 1994.

[96] Codes. Último acesso em 23/03/2013. [Online]. Disponível em: http://www.diku.dk/~pisinger/codes.html

[97] E. L. Lawler, "The quadratic assignment problem," *Management Science*, vol. 9, pp. 586–599, 1963.

[98] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the ACM*, vol. 23, pp. 555–565, 1976.

[99] S. Arora, A. Frieze, and H. Kaplan, "A new rounding procedure for the assignment problem with applications to dense graph arrangement problems," *Journal of the ACM*, vol. 23, pp. 555–565, 1976.

[100] J. Krarup and P. M. Pruzan, "Computer-aided layout design," *Mathematical Programming Study*, vol. 9, pp. 75–94, 1978.

[101] D. Taillard, "Robust taboo search for the quadratic assignment problem," *Parallel Computing*, vol. 17, pp. 443–455, 1991.

[102] N. W. Brixius and K. M. Anstreicher, "The steinberg wiring problem," in *The Sharpest Cut, Fest-Schrift in Honor of Manfred Padberg's 60th Birthday*, M. Grötschel, Ed.  SIAM, 2003, pp. 331–348.

[103] L. M. Gambardella, E. Taillard, and M. Dorigo, "Ant colonies for the quadratic assignment problem," *Journal of the Operational Research Society*, vol. 50, pp. 167–176, 1999.

[104] QAPLIB - A Quadratic Assignment Problem Library. Último acesso em 23/03/2013. [Online]. Disponível em: http://www.seas.upenn.edu/qaplib/

[105] K. S. Ghandeshtani, N. Mollai, S. M. H. Seyedkashi, and M. M. Neshati, "New simulated annealing algorithm for quadratic assignment problem," *ADVCOMP 2010*, pp. 87–92, Oct. 2010.

[106] A. Misevicius, "An implementation of the iterated tabu search algorithm for the quadratic assignment problem," *OR Spectrum*, vol. 34, pp. 665–690, 2012, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1007/s00291-011-0274-z

[107] M. Bashiri and H. Karimi, "Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons," *Journal of Industrial Engineering International*, vol. 8, pp. 1–9, 2012, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1186/2251-712X-8-6

[108] N. Fescioglu-Unver and M. M. Kokar, "Self controlling tabu search algorithm for the quadratic assignment problem," *Comput. Ind. Eng.*, vol. 60, no. 2, pp. 310–319, Mar. 2011, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/j.cie.2010.11.014

[109] Y. Xia, "An efficient continuation method for quadratic assignment problems," *Comput. Oper. Res.*, vol. 37, no. 6, pp. 1027–1032, Jun. 2010, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1016/j.cor.2009.09.002

[110] G. R. Mateus, M. G. Resende, and R. M. Silva, "Grasp with path-relinking for the generalized quadratic assignment problem," *Journal of Heuristics*, vol. 17, no. 5, pp. 527–565, Oct. 2011, Último acesso em 23/03/2013. [Online]. Disponível em: http://dx.doi.org/10.1007/s10732-010-9144-0

[111] R. Martí, M. G. Resende, and C. C. Ribeiro, "Multi-start methods for combinatorial optimization," *European Journal of Operational Research*, vol. 226, no. 1, pp. 1 – 8, 2013, Último acesso em 23/03/2013. [Online]. Disponível em: http://www.sciencedirect.com/science/article/pii/S0377221712007394