INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise de Desempenho de Técnicas de Otimização Swarm Intelligence

NELSON JOSÉ FERREIRA DA SILVA
novembro de 2016

POLITÉCNICO
DO PORTO

isep Instituto Superior de Engenharia do Porto

Instituto Superior de Engenharia do Porto ⊙ Rua Dr. António Bernardino de Almeida, 431 ⊙ 4249-015 Porto
Tel. +351 228 340 500 ⊙ Fax +351 228 321 159 ⊙ mail@isep.ipp.pt ⊙ www.isep.ipp.pt

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

isep

Análise de Desempenho de Técnicas de Otimização Swarm Intelligence

NELSON JOSÉ FERREIRA DA SILVA
Outubro de 2016

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA




Análise de Desempenho de Técnicas de Otimização Swarm Intelligence

NELSON JOSÉ FERREIRA DA SILVA
Outubro de 2016

POLITÉCNICO DO PORTO

Instituto Superior de Engenharia do Porto

Análise de Desempenho de Técnicas de Otimização *Swarm Intelligence*

Nelson José Ferreira da Silva

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientadora: Ana Madureira

Porto, outubro de 2016

Resumo

A otimização nas aplicações modernas assume um carácter fortemente interdisciplinar relacionando-se com a necessidade de integração de diferentes técnicas e paradigmas na resolução de problemas reais complexos.

Nas últimas décadas, tem havido um interesse crescente na aplicação de algoritmos que adotam de algum modo os princípios de processos estritamente naturais. São bem conhecidos os casos do Arrefecimento Simulado e da Pesquisa Tabu, permitindo a capacidade de aprendizagem e generalização. Por outro lado, a partir de paradigmas diferentes, ainda que claramente naturais, surgiram a partir de ideias iniciais de John Holland (1979) os primeiros conceitos sobre a Computação Evolucionária, introduzindo a capacidade evolutiva de adaptação e de hereditariedade de uma população.

Dentro destes tipos de Computação Evolucionária e seguindo o mesmo princípio comum - a de uma população que se autoperpetua ao longo de uma série de transformações, lutando pela própria sobrevivência - os algoritmos genéticos são de longe os mais conhecidos.

Mais recentemente, com inspiração no comportamento social dos insetos e outros animais surgiu o paradigma da Inteligência dos Enxames (*Swarm Intelligence*) desenvolvido a partir do estudo do comportamento coletivo auto-organizado de sistemas descentralizados, constituídos por um número elevado de agentes, com o objetivo de realizar uma determinada tarefa (Dorigo, 2007).

Neste contexto, esta dissertação apresenta e discute a aplicabilidade de duas técnicas de otimização com base em *Swarm Intelligence* ao problema de *Single Machine Total Weighted Tardiness,* através do desenvolvimento de uma aplicação capaz de implementar algoritmos com base nas técnicas discutidas e da realização de testes de escalamento de tarefas que permitam obter métricas relevantes para a avaliação dos resultados obtidos. Em função destes, é efetuada uma análise estatística que permite retirar conclusões sobre a análise experimental efetuada.

Este estudo permitiu concluir que o algoritmo de *Ant Colony Optimization* é capaz de obter resultados com tardeza total significativamente inferior ao algoritmo de *Artificial Bee Colony*, apesar de essa melhora se refletir num tempo de execução também maior, enquanto que o ABC é capaz de providenciar soluções muito rapidamente, a custo da qualidade da solução. A aplicabilidade de cada um dos algoritmos a este problema dependerá sempre do objetivo final da sua implementação, sendo o ACO claramente superior se o objetivo for obter a melhor solução possível. Caso o objetivo seja a obtenção de resultados com maior rapidez, então o ABC será mais adequado.

Palavras-chave: Otimização combinatória, *Swarm Intelligence*, Escalonamento de tarefas.

Abstract

The optimization in modern applications takes a strong interdisciplinary approach, relating to the integration needs of several different techniques and paradigms in solving complex real problems.

In recent decades, there has been increasing interest in the application of algorithms that take some of the principles of strictly natural processes. Cases such as Simulated Annealing and Tabu Search are well known. On the other hand, different paradigms, although clearly natural, have emerged from the initial ideas of John Holland (1979), such as the first concepts on Evolutionary Computation, which introduced the evolutionary adaptability and heredity of a population.

Within these types of Evolutionary Computation and following the same common principle - that of a population that is self-perpetuating over a series of transformations, fighting for survival - the genetic algorithms are by far the best known.

More recently, inspired by the social behavior of insects and other animals came the paradigm of intelligence of Swarms (swarm intelligence) developed from the study of self-organized collective behavior of decentralized systems, consisting of a large number of agents, with the objective to perform a certain task in an optimal form (Dorigo, 2007).

In this context, this dissertation presents and discusses the applicability of two optimization techniques based in swarm intelligence to the single machine total weighted tardiness problem, through the development of an application capable of implementing the discussed algorithms and solving appropriate task scheduling test cases, from which relevant metrics will be collected and used to properly derive the experimental analysis conclusions.

This study allows us to conclude that the Ant Colony Optimization algorithm is capable of obtaining significantly lower total tardiness values when compared to the Artificial Bee Colony algorithm, despite also requiring higher amounts of execution time, while ABC is capable of providing solutions much faster, at the cost of solution quality. The applicability of each algorithm to the mentioned problem will always depend on the implementation objectives, ACO being clearly more adequate when the objective is to obtain the best solution possible. When the objective is to obtain solutions as fast as possible, then ABC seems to be more adequate.

Key Words: Combinatorial optimization, Swarm Intelligence, Task Scheduling

Índice

1 Introdução 1

1.1 Introdução 1

1.2 Estrutura da dissertação 3

2 Definições e contexto 5

2.1 Descrição do problema 5
2.1.1 O problema de otimização 5
2.1.2 Otimização combinatória 6
2.1.3 Meta-heurísticas 8
2.1.4 Escalonamento e o seu papel 9
2.1.5 Problemas de escalonamento 10

3 Inteligência Enxame (Swarm Intelligence) 13
3.1.1 O Inseto Social e a Inteligência Enxame 13
3.1.2 Modelação de comportamento de enxames 14
3.1.3 Auto-Organização 15
3.1.4 Estigmergia 16

3.2 Algoritmos Associados 17
3.2.1 Ant Colony Optimization 17
3.2.2 Artifical Bee Colony 21
3.2.3 Particle Swarm Optimization 23
3.2.4 Differential Evolution 25
3.2.5 Stochastic Diffusiuon Search 25

3.3 Outras Aplicações 26

4 Análise experimental 29

4.1 Arquitetura da solução 29

4.2 Casos de uso e diagrama de sequência 31

4.3 Algoritmos considerados 32

4.4 Condições de teste 32

4.5 Parametrização dos casos de teste 33
4.5.2 Casos de teste 35

4.6 Detalhes da implementação 36
4.6.1 Tecnologias utilizadas 36
4.6.2 Heurística de determinação da qualidade da solução 36

5 Análise e discussão dos resultados 39

5.1 ACO com escalonamento de quarenta tarefas 39

5.2 ACO com escalonamento de cinquenta tarefas 42
5.3 ACO com escalonamento de 100 tarefas 44
5.4 ABC com escalonamento de 40 tarefas 46
5.5 ABC com escalonamento de 50 tarefas 48
5.6 ABC com escalonamento de 100 tarefas 50
5.7 Observações sobre os resultados 52

6 Conclusão 55

6.1 Trabalho Futuro 56

7 Referências 57

8 Anexos 61

8.1 Manual do utilizador 61
8.1.1 Instalação do software necessário 61
8.1.2 Parametrização da aplicação 63
8.1.3 Recolha dos dados 65
8.2 Script de SQL Server 67
8.3 Tarefas utilizadas nos escalonamentos 68
8.3.1 Escalonamentos de 40 Tarefas 68
8.3.2 Escalonamentos de 50 tarefas 69
8.3.3 Escalonamentos de 100 Tarefas 70
8.4 Código fonte da aplicação 73

Lista de Figuras

Figura 1 - Arquitetura do prótotipo 30
Figura 2 - Caso de uso do utilizador 31
Figura 3 - Diagrama de Sequência 32
Figura 4 - GetSolutionFitness() em ABC 37
Figura 5 - SolutionCost em ACO 37
Figura 6 - Qualidade da solução para 40 tarefas 41
Figura 7- Tempos de execução para 40 tarefas 41
Figura 8 - Qualidade da solução para 50 tarefas 43
Figura 9 - Tempos de execução para 50 tarefas 43
Figura 10 - Qualidade da solução para 100 tarefas 45
Figura 11 - Tempos de execução para 100 tarefas 45
Figura 12 - Qualidade da solução para 40 tarefas 47
Figura 13 - Tempos de execução para 40 tarefas 47
Figura 14 - Qualidade da solução em ABC para 50 tarefas 49
Figura 15 - Tempos de execução para ABC a 50 tarefas 49
Figura 16 - Qualidade da solução para ABC a 100 tarefas 51
Figura 17 - Tempos de execução para ABC a 100 tarefas 51
Figura 18 - P-P Plot da regressão do algoritmo ACO 52
Figura 19 P-P Plot da regressão do algoritmo ABC 53
Figura 20 - Ficheiro da solução VS 2015 62
Figura 21 - Compilação da solução 62
Figura 22 - App.Config 63
Figura 23 - Parameterização da aplicação 64
Figura 24 - Extração de dados no Excel 65
Figura 25 - Servidor SQL Server no excel 66
Figura 26 - Carregamento dos dados para Excel 66

Lista de tabelas

Tabela 1 - Algoritmo com base ACO 18
Tabela 2 - Algoritmo com base em ABC 23
Tabela 3 - Algoritmo com base em PSO 24
Tabela 4 – Casos de teste 35
Tabela 5 - Estatísticas descritivas para 40 tarefas 40
Tabela 6 - Estatísticas descritivas para 50 tarefas 42
Tabela 7 - Estatística descritiva para 100 tarefas 44
Tabela 8 - Estatística descritiva para 40 tarefas 46
Tabela 9 - Estatística descritiva para 50 tarefas 48
Tabela 10 - Estatística descritiva para ACB a 100 tarefas 50
Tabela 11 - Resultados gerais de ACO e ABC 54

Acrónimos e Símbolos

Lista de Acrónimos

SI	*Swarm Intelligence*
SDS	Stochastic Diffusion Search
ACO	Ant Colony Optimization
ABC	*Artificial Bee Colony*
CV	Caixeiro Viajante
AEM	Árvore de extensão mínima
SMTWTP	*Single Machine Total Weighted Tardiness Problem*
RCPSP	*Resource-Constrained Project Scheduling Problem*
ACS	*Ant Colony System*
AS	*Ant System*
MMAS	*Min-Max System*
PSO	*Particle Swarm Optimization*
DE	Differential Evolution

1 Introdução

1.1 Introdução

Os algoritmos de otimização por meio de inteligência enxame são uma tecnologia emergente que permite simular a evolução darwiniana e os comportamentos de seres biológicos que atuam em comunidade como uma possível solução a problemas computacionais bem conhecidos.

O conceito é inspirado no comportamento social exercido por organismos como as formigas, abelhas ou colónias de bactérias devido às suas características de inteligência emergente, com base nas ações do grupo como um todo. Embora cada elemento tenha a capacidade de comunicar direta ou indiretamente com outros elementos de forma pouco inteligente, é possível verificar comportamentos inteligentes, quando observado o comportamento gregário dos vários elementos do grupo.

Os algoritmos de otimização utilizam estas dinâmicas de grupo para tentar solucionar determinados tipos de problemas, particularmente os que ocorrem devido à computação distribuída, de modo a obter resultados satisfatórios.

As principais características deste tipo de algoritmos podem ser classificadas da seguinte forma:

- **Forte Robustez**:
    - Os algoritmos são distribuídos, o que implica que não existe nenhum tipo de controlo central. Assim, é possível aferir que um ou vários problemas em alguns dos elementos do grupo não deverá causar repercussões negativas significativas na operação geral do grupo.

- **Simplicidade**:
    - A execução e implementação de cada elemento individual do enxame é relativamente simples.
- **Escalabilidade**
    - A quantidade de informação processada por cada agente é limitada, o que permite uma melhor distribuição dos elementos que definem o problema por todos os agentes do grupo, de modo a que possam ser explorados de forma escalável.
- **Auto-organização**
    - Cada agente é responsável por si. Os comportamentos complexos do grupo são resultado das interações individuais de todos os seus elementos.
- **Homogeneidade**
    - Os elementos individuais do enxame são relativamente homogéneos, ou seja, são todos iguais ou pertencem a um reduzido conjunto de tipologias.

Os sistemas de inteligência de enxame consistem tipicamente numa população de agentes que interagem localmente entre si e entre o seu meio circundante. Os agentes seguem regras simples, e apesar da inexistência de uma estrutura de controlo centralizada, o conjunto de interações locais entre os elementos do enxame e o seu meio ambiente levam à emergência de comportamento global inteligente.

A aplicação destes princípios na indústria robótica é conhecida como robótica enxame, enquanto que a inteligência enxame refere-se a um conjunto de algoritmos de aplicação geral com as características mencionadas anteriormente.

Este tipo de algoritmos tem inúmeras aplicabilidades no mundo real, por exemplo, as primeiras aproximações à gestão de redes informáticas com base em algoritmos de enxame foram propostas em 1996 por Ruud Schoondewoed e em 1998 por Gianni Di Caro com os algoritmos *Ant-Based Control* e *AntNet* respetivamente. Este último, com performance extremamente competitiva em situações de tráfego altamente dinâmico e de caráter estocástico, como pode ser observado em redes semelhantes à Internet.

Mais recentemente, a aplicação desta tipologia de algoritmos tem sido vocacionada para áreas de estudo emergentes como o *Data Mining*, em particular com algoritmos de *Ant Colony Optimization* (ACO), a deteção de tumores através de *Stochastic Diffusion Search* (SDS), a segmentação de imagens e a deteção de patologias cerebrais através de *Artificial Bee Colony* (ABC), entre outros.

Nesta dissertação, iremos discutir a aplicabilidade destas técnicas de otimização a problemas de explosão combinatória, num contexto de escalabilidade de tarefas, onde o objetivo é obter soluções com o valor total de tardeza mais baixo possível. Um protótipo de aplicação para resolução desta classe de problemas irá ser desenvolvido, com o objetivo de permitir executar os algoritmos escolhidos num ambiente de teste pré-determinado, de modo a recolher métricas da sua execução que possam ser analisadas posteriormente em testes estáticos, que servirão de fundamentação às conclusões retiradas do estudo.

1.2 Estrutura da dissertação

Esta dissertação está dividida em sete capítulos. No capítulo II e III serão abordados todos os conceitos necessários para uma própria abordagem ao tema. No capítulo III é discutido o tema principal da dissertação, a inteligência enxame, ou *Swarm Intelligence* onde serão abordadas as dinâmicas biológicas que deram origem ao conceito, e também as principais técnicas relevantes na atualidade.

O capítulo IV abordará a parte exploratória e experimental desta dissertação. Nele está descrito um protótipo de software que terá o intuito de comparar, em termos de performance, dois algoritmos de inteligência enxame diferentes, na resolução de um problema de escalonamento real. A comparação será feita com base na qualidade das soluções conseguidas com cada um dos algoritmos, e também com base na rapidez, via análise do tempo de execução, com que os mesmos produzem soluções viáveis ao problema. Contém também uma pequena descrição de alguns dos fatores mais importantes para o desenvolvimento do protótipo, como por exemplo qual a linguagem utilizada no desenvolvimento, quais as heurísticas utilizadas no contexto do problema de escalonamento a resolver, entre outros aspetos.

No capítulo V serão apresentados os resultados dos testes e feita uma análise estatística dos mesmos, com o objetivo de extrapolar os resultados para situações reais.

Finalmente, no capítulo VI serão apresentadas as conclusões e as futuras possibilidades de continuação do percurso investigativo, com base na informação obtida na análise experimental.

2 Definições e contexto

2.1 Descrição do problema

A metáfora do inseto social como base para resolução de problemas computacionais dá ênfase a características como a capacidade de distribuição, a interação direta ou indireta entre agentes relativamente simples, a flexibilidade das implementações, a robustez, entre outros.

O crescente número de aplicações construídas com base nestes algoritmos que demonstram resultados satisfatórios tem sido evidente nos últimos anos em áreas bastante diversas como a otimização combinatória, os protocolos de comunicação e a robótica. A área da investigação tem boas razões para achar a inteligência enxame apelativa, pois numa altura em que o excesso de informação, e não a sua inexistência, ameaça o ser humano, em que o software se está a tornar tão complexo que as suas necessidades de manutenção se tornam incrementalmente mais difíceis de sustentar, este tipo de tecnologia oferece uma alternativa capaz de providenciar autonomia, emergência inteligente, e distributividade ao paradigma que temos atualmente, onde o comum é encontrar sistemas controlados, pré-programados, centralizados e com custos/necessidades de manutenção extremamente elevados.

2.1.1 O problema de otimização

Na ciência da computação o termo otimização pode ser observado como o problema de descobrir a melhor solução, num determinado conjunto de soluções possíveis, e pode ser dividido essencialmente em duas categorias dependendo de a solução ao problema ser contínua ou discreta. Um problema de otimização com variáveis discretas é conhecido como um problema de otimização combinatória e é nestes problemas onde se procura achar a melhor solução, tipicamente um número inteiro, uma permutação, ou um grafo de um conjunto de

soluções finito. Por exemplo, dado um determinado grafo G que contenha os vértices x e y um problema de otimização pode ser gerado perguntando: "Qual o caminho mais curto que vai de x até y onde tenha de passar pelo menor número possível de vértices?". Este problema pode ter uma resposta contínua, por exemplo: "Caminho 6", mas se o problema for definido com a restrição de utilizar uma travessia que utilize quatro ou menos vértices, a resposta passa a ser discreta, uma vez que a solução ao problema pode não existir ou pode existir mais do que um caminho, ou até mesmo que os caminhos possíveis não sejam os ideais. No entanto, no âmbito da otimização a solução procurada continua a ter de ser a mais eficiente ou a melhor disponível.

2.1.2 Otimização combinatória

A otimização combinatória consiste essencialmente na área de estudo da matemática e ciência computacional onde se pretende determinar o conjunto mais eficiente, dado uma determinada quantidade de conjuntos finitos.

Em muitas abordagens a este tipo de problemas, uma aproximação com base numa pesquisa exaustiva ou força bruta não é possível devido à complexidade computacional do problema. É neste domínio que operam os problemas de otimização combinatória, onde o conjunto de soluções possíveis é topologicamente discreto e o objetivo é encontrar a melhor solução possível, com resultados relativamente satisfatórios, utilizando heurísticas que aproximem as possíveis soluções daquela que é considerada a solução ótima.

Os exemplos de otimização combinatória tipicamente mais conhecidos são o problema do Caixeiro Viajante (CV) e a determinação de uma Árvore de extensão mínima (AEM) (Elsevier, 2009) .

Existem várias técnicas de obtenção de soluções aproximadas: algoritmos genéticos, redes neurais, algoritmos de inteligência enxame, nos quais esta dissertação se foca, entre outros.

O problema do caixeiro viajante consiste num vendedor e num determinado conjunto de cidades. Independentemente da cidade em que o vendedor iniciar a viagem, este deve visitar cada uma delas e retornar ao ponto inicial passando exatamente uma vez em cada cidade, fazendo o caminho mais curto possível. O vendedor sabe a distância de viajar entre uma determinada cidade *A* para qualquer outra cidade *B.*

Apesar de o problema ser de fácil descrição é computacionalmente complexo uma vez que, para qualquer solução *S,* o algoritmo de resolução ao problema deve ser capaz de determinar se existe alguma solução melhor do que *S*, o que só pode acontecer se forem analisadas todas as hipóteses possíveis. Havendo a necessidade de explorar todas as soluções para podermos afirmar que encontramos a melhor solução possível, corremos o risco de o problema se tornar num problema de resolução em tempo exponencial, pois à medida que o número de cidades aumenta, aumenta também a complexidade computacional. Estes tipos de

problemas classificam-se como sendo de complexidade NP-completo. Embora qualquer solução a um problema NP-completo possa ser validada rapidamente, não é conhecida uma forma eficiente de determinar se é a melhor solução, e no caso de não o ser, de como calcular a melhor solução possível. Mais precisamente, tal algoritmo de determinação da melhor solução existe se, e apenas se, as duas classes computacionais P e NP coincidirem, uma hipótese que com base nos últimos anos de pesquisa, é extremamente improvável. (Sanholzer, 1991).

De um ponto de vista prático, isto significa que é impossível encontrar um algoritmo exato para qualquer instância deste problema, com *n* cidades, em que *n* é um número relativamente elevado, que tenha um comportamento consideravelmente melhor que o algoritmo que calcule qualquer uma das (*n-1*)! hipóteses e retorne a mais eficiente.

Algoritmos que consigam construir soluções em tempo polinomial, relativamente às *n* soluções possíveis, e aproximar à solução ideal, dizem-se algoritmos com base heurística.

Em geral, estes algoritmos não oferecem garantia de que uma determinada solução está relativamente próxima da solução ideal, pois apenas é possível demonstrar que o resultado de uma solução é *k* vezes melhor do que um qualquer resultado anterior, para um número real de $k > 1$. A heurística associada é chamada de $k - approximation$ nestes casos.

Infelizmente, algoritmos de $k - approximation$ não são conhecidos para problemas de caixeiro viajante, o que torna a definição de uma boa heurística extremamente difícil.

É neste contexto que aparecem várias técnicas de heurística, onde se incluem os algoritmos de inteligência enxame, que têm como objetivo produzir resultados satisfatórios, em tempo útil. As técnicas modernas conseguem encontrar soluções relativamente rápido, e estatisticamente próximas da melhor solução.

Existem várias categorias de heurística construtiva para este problema específico, das quais destacamos duas:

- ***Nearest Neighbour***: Uma das primeiras heurísticas a ser utilizada para resolver o problema de CV. Funciona escolhendo uma cidade aleatória e, recursivamente, visita a cidade mais próxima caso esta ainda não tenha sido visitada, até percorrer as cidades todas. Esta heurística dá uma solução ao problema rapidamente, mas usualmente não a melhor.
- ***Match Twice And Stitch***: escolhendo uma cidade, o vendedor faz corresponder [1] sequencialmente duas cidades, onde a segunda correspondência é feita após excluir todos os caminhos da primeira correspondência, de modo a construir um conjunto de ciclos, que são unidos no final dando resultado ao caminho a seguir (solução). (Bentley, 1992)

[1] Via *Matching* – Teoria de grafos

2.1.3 Meta-heurísticas

Enquanto os algoritmos de heurística construtiva mencionados anteriormente são criados com um propósito específico de aplicação, e, portanto, aplicáveis também a problemas de caráter particular. As meta-heurísticas são heurísticas de alto nível, independentes de problemas específicos, que providenciam um conjunto de regras e/ou estratégias a seguir para desenvolver algoritmos de otimização. Exemplos notáveis de meta-heurísticas incluem algoritmos genéticos, pesquisa tabu, arrefecimento simulado, *Variable Neighbourhood Search* (VNS), *Large Neighbourhood Search* (LNS), *e Ant Colony Optimization* (ACO), apesar de existirem bastantes mais. Casos como a ACO serão discutidos no capítulo de *Swarm Intelligence* desta dissertação.

Uma implementação específica de um algoritmo de otimização heurística de acordo com as regras definidas numa *framework* meta-heurística é também denominado como meta-heurística. O termo foi introduzido por Glover (Glover, 1986).

Algoritmos meta-heurísticos são, como o nome sugere, sempre heurísticos na sua natureza. Este facto distingue-os de métodos exatos, normalmente caracterizados por fornecerem soluções ideais, em quantidades de tempo finitas, apesar de proibitivamente grandes para determinados casos. Meta-heurísticas surgem como alternativa viável para determinar soluções suficientemente boas em tempo útil, e não são, portanto, sujeitos ao fenómeno de explosão combinatória, denominado por requererem quantidades de tempo de resolução crescentes de forma exponencial, de acordo com a complexidade do problema a resolver.

Para problemas de dificuldade extrema, ou de tamanho considerável, a meta-heurística é frequentemente a alternativa que oferece a melhor relação qualidade/tempo de execução. Para além disso, são mais flexíveis do que os métodos exatos em duas características importantes: primeiro, uma vez que uma meta-heurística é definida em termos genéricos, os algoritmos podem ser adaptados de modo a preencher as necessidades específicas da grande maioria dos problemas de otimização; segundo, a meta-heurística não cria dependências na formulação do problema de otimização, apesar desta flexibilidade incluir a desvantagem de necessitar de bastante adaptação ao problema de modo a atingir parâmetros de performance aceitáveis.

A capacidade de obter boas soluções em casos onde outros métodos falham tornou a meta-heurística na melhor escolha para resolver a maioria dos problemas de otimização da vida real, tanto na investigação científica como em aplicações práticas, o que fez com que vários vendedores de software implementassem meta-heurísticas como motores de otimização geral nas suas aplicações, bem como em aplicações de caráter industrial.

Os algoritmos de meta-heurística tentam encontrar a melhor solução viável de entre todas as possíveis soluções de um problema de otimização. Para este fim, avaliam potenciais hipóteses e executam uma série de operações, de modo a determinar outras hipóteses de solução, ou soluções melhores. Para tal, mantêm em memória um objeto representativo da

solução que possa ser convenientemente manipulado pelos agentes do algoritmo. Três classes de meta-heurística podem ser identificadas através da forma como a manipulação do objeto é executada:

- ***Local Search*:** pesquisa local, também conhecida como melhoria iterativa, encontra boas soluções fazendo pequena alterações, de forma iterativa, numa única solução inicial.
- ***Constructive*:** a meta-heurística construtiva, tal como o nome sugere, constrói soluções através dos elementos constituintes, e não através da melhoria de uma solução inicial, adicionando um elemento da solução de cada vez a uma solução parcial.
- ***Population Based*:** esta meta-heurística constrói soluções selecionando e combinando soluções existentes encontradas dentro de um conjunto (ou população).

A maioria dos algoritmos de meta-heurística criados recentemente combinam de alguma forma estas três classes, com o objetivo de encontrar as melhores soluções possíveis, sendo considerados de modo coloquial como algoritmos de meta-heurística híbrida.

2.1.4 Escalonamento e o seu papel

O escalonamento é a atividade de tomar decisões, definir ordem e atribuir tarefas. Quando aplicado à área da informática, este termo geralmente refere-se à atribuição de elementos computacionais (processos, *threads*) a recursos físicos (processador, disco) limitados, durante um determinado espaço de tempo, de modo a estes poderem executar as suas tarefas.

Numa organização, os recursos podem ser máquinas de uma fábrica, pistas de descolagem num aeroporto, equipas num local de construção, unidades de processamento num sistema de computação, etc. Tipicamente as tarefas a ser executadas têm algum tipo de prioridade, uma data de início e uma data de fim, de onde é possível extrair então a quantidade de tempo que necessitarão de um determinado recurso para completar a tarefa.

Os objetivos do escalonamento podem ser variados, podendo por vezes pretender-se garantir que nenhuma das tarefas é completada após a sua data de fim, ou, apenas organizar as tarefas de modo a que seja feito o máximo de trabalho possível no menor curto espaço de tempo. Estes objetivos podem, ou não, ser compatíveis dependendo do problema, mas, tipicamente o processo de escalonamento é definido inicialmente com uma heurística pré-determinada, para resolver um tipo de problema específico.

A título de exemplo, iremos analisar o processo de escalonamento de tarefas numa unidade de processamento central de um computador. Uma das funções de um sistema operativo é precisamente escalonar a quantidade de processador atribuindo a cada um dos diferentes programas que estão em execução num dado momento, jogando com as prioridades das tarefas e os seus tempos de execução necessários, de modo a minimizar o tempo de espera de realização de todas as tarefas que estejam a correr no sistema.

De modo a reduzir situações em que tarefas de baixa prioridade e baixo tempo de execução fiquem indefinidamente à espera de ser executadas, visto poderem existir outras tarefas mais urgentes e que requerem mais recursos, o processador tenta separar as tarefas em tarefas mais pequenas e inseri-las no escalonamento, de modo a que num determinado espaço de tempo, o processador tenha pelo menos gasto algum tempo, por muito pouco que seja, a executar esses pedaços da tarefa, até que esteja completa e possa sair rapidamente da fila de espera do sistema.

O escalonamento pode ser difícil de um ponto de vista técnico, mas também de um ponto de vista da implementação, uma vez que as suas dificuldades técnicas são bastante semelhantes às de qualquer outro problema de otimização combinatória ou de modelação estocástica. A performance de um qualquer algoritmo de escalonamento será sempre dependente da dificuldade das tarefas a escalonar e do modelo analítico que define qual a prioridade do escalonamento, algo que depende obviamente do seu objetivo. (Pinedo, 2002)

Existem vários tipos de software no mundo real que necessitam de resolver problemas de escalonamento inerentemente distribuídos. Entre estas aplicações incluem-se sistemas de resgate, sistemas de melhoria de rede, software de gestão de projeto, sistemas de escalonamento de horário e redes de sensores. Eles exigem soluções distribuídas que sejam capazes de atingir resultados confiáveis e flexíveis. À medida que cenários de larga escala, com centenas de recursos e tarefas se tornam ubíquos, novos desafios aparecem. Soluções ótimas a este tipo de problema não são possíveis de atingir em sistemas de larga escala, com computação e escalonamento distribuído, devido às necessidades computacionais e de comunicação associados com a pesquisa da solução ideal. Por outro lado, soluções aproximadas que são capazes de lidar com este tipo de problemas permanecessem por investigar.

2.1.5 Problemas de escalonamento

2.1.5.1 Single Machine Total Weighted Tardiness Problem (SMTWTP)

O problema de escalonamento de atraso pesado em máquina única, ou *Single Machine Total Weighted Tardiness Problem* (SMTWTP) pode ser descrito da seguinte forma: para cada *n* tarefas (1…*n*), cada uma deve ser processada não-interruptamente numa máquina que deve ter

a capacidade de processar apenas uma tarefa de cada vez. Cada tarefa *j* (j = 1…n) torna-se disponível para processamento no instante de tempo zero, requer processamento positivo e não interrupto p_j, tem um peso positivo w_j, e uma data de fim d_j, correspondente ao limite de tempo em que a tarefa *j* idealmente terá sido completada. Para um determinado conjunto de tarefas, um tempo de conclusão C_j, e um atraso $T_{j=\max\{C_j - d_j,\ 0\}}$ cada tarefa *j* pode ser efetivamente computada. O problema consiste em encontrar a ordem de processamento das tarefas de modo a que estas sejam executadas com o mínimo total de atraso considerado $\sum_{j=1}^{n} w_j T_j$.

Este problema de escalonamento não é apenas de complexidade NP, mas também difícil de um ponto de vista prático, visto que o algoritmo de Potts e Van Wassenhove (1985), considerado o mais atual e eficiente, tem severos problemas em conseguir escalonar instâncias com mais de 50 tarefas. Resultados aproximados são também extremamente difíceis de obter, uma vez que os algoritmos existentes não garantem resultados aproximados à solução ideal em tempo polinomial (Potts & Wassenhove, 1985).

2.1.5.2 Resource-Constrained Project Scheduling Problem (RCPSP)

O problema de escalonamento de projeto com recursos limitados, ou *Resource-Constrained Project Scheduling Problem* (RCPSP) é um problema de escalonamento de caráter geral que pode ser utilizado para modelar inúmeras atividades, como por exemplo, um processo de produção, um projeto de software, um calendário escolar, a construção de uma casa, entre outros. O objetivo é escalonar tarefas ao longo de um determinado período de tempo de modo a otimizar a utilização de recursos escassos, e de modo a cumprir com o objetivo do escalonamento também de forma otimizada. Exemplos de recursos podem ser maquinaria, pessoas, dinheiro, energia, que se encontram disponíveis em quantidades limitadas. Funções do escalonamento podem incluir a duração total, o desvio de cumprimento dos prazos das tarefas, ou o custo, todas estas podendo ser otimizadas.

O RCPSPS pode ser formalizado da seguinte forma: dado um determinado horizonte temporal $[0, T]$, *n* tarefas (1…n), e *r* recursos renováveis (1…*r*). Uma quantidade constante de R_k unidades de recurso *k* a qualquer momento de *T*. A tarefa *i* deve de processada em p_i unidades de tempo. Durante este período, uma quantidade constante de r_{ik} unidades de recurso é ocupada.

Além do mais, constrangimentos precedentes são definidos entre as tarefas, sendo derivadas através de uma relação *i->J*, onde *i->j* significa que a tarefa *j* não pode ser iniciada antes de a tarefa *i* ter sido terminada com sucesso.

O objetivo é determinar tempos de início para todas as tarefas de modo a garantir que:

- Em qualquer momento a quantidade de recursos necessários para processar as tarefas é inferior à sua disponibilidade;
- Os constrangimentos de precedência são satisfeitos, $S_i + p_i \leq S_j \; se \; i \; precede \; j$;
- O período de tempo necessário para executar as tarefas é o mais curto possível, de modo a que $C_{max} = max_{i=1}^{n} \{C_i\}$ seja minimizado.

O vetor $S = (S_i)_{i=1}^{n}$ define o escalonamento do projeto. S é considerado viável se todos os recursos e precedentes de escalonamento forem cumpridos. Se o horizonte temporal *T* for suficientemente largo, os requisitos de precedência forem cumpridos e se $r_{ik} \leq R_k$ for sempre válido, a viabilidade do escalonamento fica assegurada.

Neste capitulo estabelecemos qual o problema que pretendemos resolver ao aplicar técnicas de *swarm intelligence* bem como a sua importância no contexto da otimização combinatória. O escalonamento de tarefas é, como foi demonstrado de forma clara, uma classe de problemas que exige elevadas quantidades de investigação, visto a sua eficiência ser um componente absolutamente fundamental em muitas das suas aplicações no mundo real.

3 Inteligência Enxame (*Swarm Intelligence*)

A inteligência enxame é o ramo das ciências computacionais que desenha e estuda métodos de computação eficientes, com o objetivo de resolver problemas de computação complexos, inspirando o procedimento dos algoritmos no comportamento observado em seres biológicos, como enxames de abelhas ou colónias de insetos. Princípios de auto-organização e comunicação direta ou indireta são importantes para compreender o comportamento coletivo complexo destes seres biológicos. Exemplos onde conceitos extraídos destes comportamentos influenciaram algoritmos e sistemas na ciência computacional incluem:

- O transporte coletivo de alimento nas formigas inspirou o desenho de robôs cujo objetivo é executar trabalho coletivamente (Kube & Bonabeau, 2000);
- A triagem coletiva executada pelas formigas motivou vários algoritmos de *sorting* e *clustering* (Meyer & Handl, 2002), (Faieta & Lumer, 1994);
- As capacidades de orientação e de encontrar caminhos da formiga do deserto *Cataglyphis* foram usadas como modelo para a construção de uma unidade de orientação robótica (Lambrinos, 2000).

Neste capítulo, procura dar-se o devido contexto aos elementos e comportamentos que constituem o interesse da ciência da computação neste ramo, explorando algumas das soluções existentes e explicando os comportamentos biológicos que o influenciaram.

3.1.1 O Inseto Social e a Inteligência Enxame

É possível observar nos insetos que vivem em colónias, como as abelhas e as formigas, padrões de comportamento. Cada inseto aparenta ter a sua agenda individual no contexto da

colónia, mas para o observador é impossível ignorar a aparente harmonia e organização da mesma, apesar da inexistência de um elemento supervisor.

Um inseto é uma criatura relativamente complexa, podendo processar inúmeras experiências sensoriais, modular o seu comportamento de acordo com estímulos, incluindo interações com outros elementos da colónia, e tomar decisões com base de acordo com a informação recolhida. No entanto, a complexidade de cada elemento da colónia continua a não ser suficiente de modo a explicar a complexidade das ações da colónia. De onde surge a cooperação?

Alguns dos mecanismos por trás da cooperação são determinados por questões de genética, como as diferenças anatómicas dos indivíduos, que podem determinar a distribuição do trabalho. No entanto, muitos dos aspetos das atividades coletivas da colónia são organizados entre si próprios, o que permite observar a emergência de determinados micro padrões, podendo assim explicar o modo como determinados comportamentos complexos coletivos podem emergir, através de interações e comportamentos relativamente simples entre os elementos individuais. Nestes casos não existe a necessidade de identificar complexidade individual para explicar a complexidade dos comportamentos coletivos.

Estas características de auto-organização permitem definir modelos que não precedem necessariamente de complexidade individual, permitindo assim explicar comportamentos complexos assumindo que cada elemento individual necessite apenas de capacidades básicas de interação dentro do sistema (Dorigo, et al., 1999).

3.1.2 Modelação de comportamento de enxames

Umas das necessidades inerentes à criação de sistemas de inteligência artificial e algoritmos capazes de resolver problemas, com base em comportamentos inspirados nas interações sociais entre os insetos, é a modelação. A modelação é substancialmente diferente da fase de design do sistema, uma vez que aqui pretende-se entender e descrever o que acontece no sistema natural que serve de inspiração.

Não só deve o modelo reproduzir o que acontece no sistema natural que é suposto descrever, como deve ser consistente com o que é conhecido do sistema natural que tenta simular. Isto é, valores de parametrização de componentes do sistema não devem ser arbitrais, e os mecanismos do modelo devem ter aplicabilidade ao sistema biológico.

3.1.3 Auto-Organização

Auto-organização é um conjunto de mecanismos dinâmicos que se manifestam ao nível global de um sistema, através das interações entre os componentes de baixo-nível. As regras que especificam as interações dos elementos constituintes do sistema são executadas com base na informação local de cada um, sem qualquer tipo de referência ao padrão global do sistema, manifestando-se neste de forma emergente, e não por imposição de forças externas (Dorigo, et al., 1999).

Por exemplo, a estrutura emergente no comportamento de uma colónia de formigas cujos elementos estejam em busca de alimento deve-se, em parte significativa, à existência de redes espácio-temporais compostas por diferentes quantidades de feromonas, que influenciam o comportamento individual de cada formiga.

A auto-organização baseia-se essencialmente em quatro elementos. O reforço positivo (também conhecido como amplificação) é o principal elemento constituinte e que serve como base da morfogénese do sistema. Consiste essencialmente em regras simples de comportamento que promovem a criação de estrutura no sistema. Por exemplo, encontrar uma fonte de alimento é um reforço positivo promovido pela criação e seguimento do rasto de feromonas deixado pelas formigas, ou um tipo de dança no caso das abelhas. Quando uma abelha encontra uma fonte de néctar, regressa à colónia para informar os restantes elementos. Esta pode então começar a dançar para indicar às outras abelhas a direção e distância da fonte de alimento, continuar a perspetivar a fonte sem recrutar outros elementos da colónia, ou pode abandoná-la e tornar-se numa abelha seguidora de outra fonte, caso esta exista. Se a colónia dispuser de várias fontes de néctar pode explorá-las de forma simétrica, no entanto, se uma das fontes for claramente melhor do que outra, a colónia tem a capacidade de explorar a melhor fonte imediatamente, ou quando for mais conveniente.

O reforço negativo contrabalança o reforço positivo e ajuda o padrão coletivo a estabilizar podendo tomar como manifestação a saturação, a exaustão ou a competição. No exemplo de comportamento exploratório das formigas, o reforço negativo manifesta-se através do limite de membros exploratórios da colónia, da saciedade, da exaustão dos elementos exploratórios, do excesso dos indivíduos a explorar as fontes de alimento, e da competição pelos recursos quando estes são escassos.

A auto-organização depende da amplificação ou flutuação (tentativas de exploração aleatórias, mudança repentina de tarefa que o elemento está a executar, erros, etc.). Não só as estruturas emergem apesar da aleatoriedade, mas esta é crucial para o bom funcionamento da colónia, já que permite a descoberta de novas soluções, e as flutuações podem agir como ponto inicial da emergência de estruturas e do seu crescimento. Por exemplo, no cenário de uma formiga exploratória, esta pode efetivamente encontrar-se perdida por ter seguido um rasto de feromonas de pouca qualidade, o que apesar de parecer um fenómeno que gera ineficiência, pode permitir à formiga perdida encontrar novas fontes de recursos, e recrutar outros elementos para ajudar na exploração dos mesmos.

Todos os casos de auto-organização acontecem apenas devido às inúmeras interações entre os elementos. Um elemento individual pode efetivamente gerar um rasto de feromonas estável e eficiente, mas devido ao tempo útil de vida da feromona, pode não ser suficiente para influenciar o comportamento da colónia. Para tal, é essencial uma densidade relativamente significativa de indivíduos, já que o comportamento de seguir o rasto de feromona não é incompatível com o comportamento de criar o rasto de feromona, e quanto maior a densidade desta num determinado rasto, maior a probabilidade de outros elementos da colónia a seguirem. No entanto, a capacidade dos diferentes elementos de detetar e seguir a feromona existente, não os impede de identificar a sua própria assinatura química, e de a utilizar para complementar ou por vezes até substituir a assinatura coletiva. (Dorigo, et al., 1999)

3.1.4 Estigmergia

A auto-organização nos insetos sociais requer interações, podendo estas ser diretas ou indiretas. Interações diretas são frequentemente as mais aparentes – troca de comida ou líquidos, contacto mandibular, contacto visual ou contacto químico (odor, feromona) etc.

Interações indiretas são tipicamente mais subtis, ocorrendo quando um elemento da colónia altera o ambiente, alteração essa que irá causar algum tipo de resposta em algum outro elemento. Este tipo de interações são exemplos de estigmergia. Em adição a, ou em conjunto com a auto-organização, a estigmergia é um outro componente importante do funcionamento das colónias. Grassé introduziu o conceito para explicar a coordenação de tarefas e regulação das mesmas no contexto da construção de ninhos de térmitas (Grassé, 1959) e demonstrou que a coordenação e regulação das diferentes tarefas não dependem dos elementos construtores individualmente, mas sim da estrutura do próprio ninho: uma determinada configuração desencadeia uma resposta na térmita construtora, transformando a estrutura do ninho numa outra configuração que pode desencadear uma outra resposta (possivelmente diferente) na própria térmita, ou em qualquer outra térmita na colónia.

A estigmergia de um determinado sistema pode ser facilmente negligenciada, uma vez que não explica detalhadamente os mecanismos de coordenação das atividades dos elementos da colónia. No entanto, permite subentender e relacionar mecanismos que coordenam o comportamento dos indivíduos da colónia com o comportamento geral: um elemento muda o ambiente, o que por sua vez pode mudar o comportamento de outros elementos da colónia.

O caso da construção do ninho de térmitas demonstrado por Grassé é um bom exemplo de como a estigmergia pode ser utilizada para coordenar as atividades de construção das térmitas, por meio de auto-organização. (Dorigo, et al., 1999).

Outro exemplo de estigmergia é a maneira como a alta performance demonstrada pelos elementos da colónia numa determinada tarefa determina a necessidade de menos performance nos outros elementos: por exemplo, uma alta performance na limpeza do ninho

reduz a necessidade de continuidade dessa tarefa, ou seja, alguns elementos da colónia comunicam entre si modificando o ambiente, por via de limpeza do ninho, e outros elementos respondem ao ambiente modificado parando de limpar o ninho e vocacionando as suas capacidades para outras tarefas.

Também relevante como observação de estigmergia é a maneira como as térmitas utilizam a construção de outras térmitas como base das suas próprias criações, ou seja, construções incrementais, o que contextualizado à otimização de algoritmos é extremamente relevante, pois vários dos algoritmos utilizam melhorias incrementais como base da sua operação – uma nova solução é construída com base nas soluções anteriores.

A estigmergia é também por vezes associada à flexibilidade: quando o ambiente muda devido a perturbações exteriores, os elementos da colónia respondem de maneira adequada a essa perturbação, como se fosse uma modificação do ambiente causado pelas atividades da colónia. Quando se tratam de agentes artificiais, este tipo de flexibilidade é bastante valioso, pois significa que os agentes têm a capacidade de responder a uma perturbação do seu ambiente sem necessidade de reprogramar os mesmos para lidar com a perturbação.

Estes exemplos demonstram que a estigmergia pode rapidamente tornar-se operacional dentro deste tipo de sistemas.

3.2 Algoritmos Associados

3.2.1 Ant Colony Optimization

A Ant Colony Optimization (ACO) tem como inspiração o comportamento exploratório de algumas espécies de formigas.

Estas, durante o seu esforço exploratório, depositam feromonas nas superfícies de modo a marcar caminhos favoráveis a ser seguidos pelos restantes elementos da colónia.

Na ACO, um mecanismo semelhante é explorado para resolver problemas de otimização, tendo sido proposto inicialmente por Marco Dorigo (Dorigo, 1992) na sua tese de doutoramento.

No mundo natural, quando uma formiga sai da colónia para explorar novas fontes de alimentos, esta começa por explorar o ambiente circundante de modo aleatório, e ao encontrar uma fonte de alimento, retorna à colónia deixando um rasto de feromonas. Se outra formiga encontrar esse rasto, existe a possibilidade de esta deixar de procurar alimento também de forma aleatória, e em vez disso, passar a seguir o rasto deixado pela formiga anterior, reforçando assim o rasto de feromonas, se esta também encontrar a fonte de alimento.

Eventualmente, quando a fonte de alimento se esgotar, e as formigas deixarem de sustentar o rasto de feromonas, esta tende a evaporar-se, reduzindo assim o seu efeito nos restantes elementos da colónia, e fazendo com que as restantes formigas deixem de seguir o rasto até à fonte de alimento extinta. Por outro lado, se um caminho para outra fonte mais próxima for encontrado, este é provavelmente usado com mais frequência do que o anterior, o que aumenta a sua densidade de feromonas, fazendo com que a colónia passe a utilizar este caminho em substituição do anterior. Este é o efeito positivo da evaporação temporal das feromonas, visto que quando uma formiga encontra um caminho mais otimizado para uma fonte de alimento, através do *feedback* positivo e da evaporação de feromonas do caminho anterior, eventualmente todas as formigas da colónia irão optar pelo caminho otimizado.

A teoria por detrás de um algoritmo com base na ACO é de certa forma imitar este tipo de comportamento com formigas artificiais (agentes) que atravessam grafos, que representam o problema a resolver. Estas formigas artificiais, constroem soluções de forma incremental à medida que se movimentam no grafo. O processo de construção das soluções é estocástico e baseado no modelo das feromonas, ou seja, os parâmetros associados com os componentes do grafo são modificados pelas formigas artificiais durante a execução à medida que estas atravessam o grafo.

Um exemplo relativamente simples de um algoritmo baseado em ACO, pode ser visualizado na tabela 1.

```
Input: Parâmetros ACO e problema de escalonamento
Ouput: Melhor Solução
Begin
    Atribuir parâmetros ACO
    Inicializar rasto de feromona
    While critério de fim não satisfeito do
        Contruir soluções das formigas
        Aplicar pesquisa local (opcional)
        Atualizar feromona
        Memorizar a melhor solução existente até ao momento
    End
End
```

Tabela 1 - Algoritmo com base ACO

De seguida, apresentam-se os principais algoritmos associados a soluções de ACO.

3.2.1.1 Ant System

Ant System foi o primeiro algoritmo com base em ACO a ser proposto na literatura por Marco Dorigo (Dorigo, 1992). A sua principal característica é que os valores das feromonas são atualizados por todos os agentes que tenham percorrido o grafo em cada iteração. A feromona T_{ij}, é atualizada segundo a equação:

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m}\Delta\tau_{ij}^{k}$$

Onde ρ é o rácio de evaporação, *m* é o número de formigas (agentes), e T_{ij}^{k} é a quantidade de feromona largada no vértice *(i, j)* do grafo, pela formiga *K:*

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{L_k} & \text{se k usa o vértice (i, j)} \\ 0 & \text{se não} \end{cases}$$

Onde Q é a constante, e L_k é o tamanho do caminho construído pela formiga *k*.

Na construção de uma solução, as formigas selecionam o vértice seguinte a visitar de forma estocástica. Quando uma formiga *k* está no vértice *i* e até ao momento construiu uma solução parcial S_k^p, a probabilidade da formiga seguir para o vértice *j* é dada por:

$$p(c_{ij}|s_k^p) = \begin{cases} \dfrac{\tau_{ij}^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{c_{il}\in N(s_k^p)}\tau_{il}^{\alpha}\cdot\eta_{il}^{\beta}} & \text{se } \; j \in N(S_k^p) \\ 0 & \text{se não} \end{cases}$$

Onde $N(S_k^p)$ são os vértices que ainda não fazem parte da solução parcial S_k^p de uma formiga *k, e α e β* são parâmetros que controlam a importância da feromona *versus* a informação heurística n_{ij}, que é dada pela equação $n_{ij} = \frac{1}{d_{ij}}$ onde d_{ij} é a distância entre *i* e *j.*

3.2.1.2 Ant Colony System

Ant Colony System (ACS) é o primeiro algoritmo criado especificamente como melhoria do *Ant System* (AS) original, introduzido por Dorigo e Gambardella (1997). A principal diferença entre ACS e o AS é a formula da regra de decisão utilizada pelas formigas durante o processo de construção da solução. No ACS, as formigas utilizam uma regra pseudoaleatória – a probabilidade de uma formiga de escolher ir para *i* ou *j* depende de uma variável *q* distribuída uniformemente entre [0, 1], e um parâmetro q_0. Se $q \leq q_0$, o componente que maximize $T_{il}n_{il}^{\beta}$ é escolhido. Caso contrário, a mesma formula de AS é utilizada.

Esta regra, que dá preferência à exploração da informação da feromona, é contrabalançada pela introdução de um componente diversificador: a atualização da feromona local. Esta é executada por todas as formigas após cada passo na construção da solução e aplica-se apenas ao último passo executado:

$$\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0$$

Onde $\alpha \in [0, 1]$ é o coeficiente de decomposição da feromona, e T_0 é o valor inicial da feromona.

A atualização da feromona local tem como objetivo diversificar a exploração das formigas que atravessem um determinado vértice subsequentemente durante uma iteração do algoritmo. Esta modificação encoraja as formigas subsequentes a escolher outros caminhos que não o travessado pela formiga anterior e, portanto, a procurar outras soluções para o problema, o que tem como objetivo evitar que um elevado número de formigas produza o mesmo resultado para o mesmo problema.

Tal com no AS, no ACS é executada uma atualização à feromona no final de cada processo de construção de soluções, a diferença é que no ACS, apenas a formiga com a melhor solução executa esta atualização e a atualização é executada apenas nos locais atravessados apenas por esta formiga, de acordo com a seguinte equação:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij}^{best}$$

Onde $\Delta\tau_{ij}^{best}$ $\frac{1}{L_{best}}$ caso a formiga tenha utilizado o traço (*i, j*) na sua solução e L_{best} pode ser o tamanho do melhor caminho encontrado na iteração corrente (*iteration best*, ou L_{ib}), ou o tamanho do melhor caminho encontrado desde o inicio do algoritmo (*best so far*, ou L_{bs}).

3.2.1.3 Max-Min Ant System

O sistema *Max-Min Ant System* (MMAS) é outra melhoria, proposta por Stützle e Hoos (2000), para o *Ant System*. O MMAS difere do AS da seguinte forma:

- Apenas a melhor formiga deixa rasto de feromona;
- Os valores máximos e mínimo da feromona ($T_{min} - T_{max}$) são explicitamente limitados, ao contrário do AS e ACS, onde os limites são determinados pela execução do algoritmo.

A equação de atualização da feromona é a seguinte:

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \Delta\tau_{ij}^{best}$$

Onde $\frac{1}{L_{best}}$ caso a formiga tenha utilizado o traço (*i, j*) na sua solução e L_{best} é o tamanho do caminho da melhor formiga. Tal como em ACS, L_{best} pode ser L_{ib} ou L_{bs}, ou uma combinação de ambos. No entanto, o nível de feromona é sempre constrangido aos valores entre T_{min} e T_{max} após a atualização feita pelas formigas (Stutzle & Hoos, 1997).

3.2.2 Artifical Bee Colony

Algoritmos baseados em *Artifical Bee Colony* (ABC) são dos mais recentes na área de inteligência enxame tendo sido propostos por Pham em 2005 (Pham, 2005) e Karaboga (Karaboga & Akay, 2011) com o objetivo de simular o comportamento dos enxames de abelhas na procura de fontes de alimento. A comunicação entre as abelhas alerta as mesmas não só para a direção da fonte de alimento, mas também para as quantidades de néctar disponíveis. Esta troca de informação ajuda as abelhas a detetar as fontes de alimento mais otimizadas, em benefício da colmeia.

Nos algoritmos com base em ABC, este comportamento coletivo das abelhas é simulado com o objetivo de otimizar sobretudo soluções de problemas de pesquisa, mas tem sido também aplicado na otimização de vários tipos de problemas (Karaboga, 2012), (Sharma & Bansal, 2012).

As colónias de abelhas artificiais tipicamente são constituídas por três tipos de abelhas: a abelha obreira, a abelha observadora e a abelha exploradora. As abelhas obreiras existem na mesma quantidade que as fontes de alimento (soluções ao problema) e são responsáveis pela exploração de uma, e apenas uma, das fontes de alimento, e de transmitir a informação às restantes abelhas. As abelhas observadoras esperam na colmeia pela informação recolhida pelas abelhas obreiras e são responsáveis por tomar a decisão de qual a fonte de alimento a ser explorada. As abelhas exploradoras exploram o ambiente circundante de modo a encontrar novo alimento, com base nas suas motivações internas, pistas visuais, ou de forma de aleatória. De notar que, neste tipo de algoritmos, o agente que representa a abelha obreira tipicamente torna-se numa abelha exploratória assim que deixa uma fonte de alimento, tendo como vantagem a capacidade de abranger o alcance e/ou profundidade da pesquisa.

A localização de uma fonte de alimento representa uma possível solução ao problema de otimização, e a quantidade de néctar dessa fonte corresponde à qualidade da solução associada. Neste algoritmo, metade da colmeia consiste em abelhas obreiras artificiais e a outra metade em abelhas observadoras, sendo o número de ambas igual à quantidade de soluções existentes, tal como referido anteriormente.

Na fase de inicialização, o algoritmo gera aleatoriamente $\frac{sn}{2}$ soluções, onde *sn* é o tamanho da população de abelhas, e onde o número de solução seja igual ao número de abelhas obreiras. A cada x_i (*i =1, 2, sn/2*) é um vetor dimensional *D*. Valores entre os limites da parametrização são atribuídos à solução e um valor de $falha_i$ é também adicionado para os casos em que uma solução *i* deve ser abandonada.

Após validação da população de abelhas, o algoritmo repete um determinado número de ciclos para cada tipo de abelha.

3.2.2.1 Abelha obreira

A abelha obreira produz uma modificação na localização (solução) que tem em memória, dependendo da informação visual local, e posteriormente testa a quantidade de néctar da nova localização (nova solução). Presumindo que a concentração de néctar da nova localização é maior do que a anterior, a abelha memoriza a nova posição e esquece a primeira, caso contrário, mantém a informação que tinha. Após todas as abelhas obreiras completarem a fase de pesquisa, estas partilham a informação com a abelha observadora na colmeia.

3.2.2.2 Abelha observadora

A abelha observadora avalia a informação do néctar recolhida por todas as obreiras e escolhe qual a localização com néctar de melhor qualidade. Tal como no caso da abelha obreira, produz uma modificação na sua memória e valida a qualidade do néctar na localização que lhe foi partilhada, o que no caso desta conter néctar melhor, faz com que a abelha observadora memorize a nova localização (solução) e esqueça a anterior.

3.2.2.3 Abelha exploradora

Após os passos anteriores, todas as fontes de alimento que não serão exploradas são abandonadas. A abelha obreira que estava atribuída a uma das localizações abandonadas, passar a procurar por fontes de alimento de forma aleatória. Em situações normais as abelhas exploradoras não correspondem a mais de 10% da população total de abelhas.

Seguidamente, na tabela 2, é possível visualizar um algoritmo de ABC.

Input: Parâmetros ABC e problema de escalonamento

Ouput: Melhor Solução

Begin

Inicializar população de abelhas e fontes de alimento

ciclo = 1

While ciclo <> número máximo de ciclos **do**

Avaliar soluções

Fase da abelha obreira

Calcular probabilidades com base nas abelhas observadoras

Fase da abelha observadora

Fase da abelha exploratória

Memorizar a melhor solução alcançada até ao momento

Incrementar ciclo

End

End

Tabela 2 - Algoritmo com base em ABC

3.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) é uma técnica de otimização estocástica, contínua, e discreta, com base em populações de agentes (partículas).

Na PSO, agentes relativamente simples, movem-se num determinado espaço de pesquisa com o objetivo de encontrar uma otimização para um determinado problema. A posição de cada partícula representa uma possível solução para o problema de otimização que está a decorrer durante a execução do algoritmo. Ao movimentarem-se, cada partícula procura por uma posição no espaço de pesquisa, mudando a sua velocidade de acordo com regras originalmente inspiradas no modelo comportamental de determinadas espécies de aves que atuam em bando, e/ou em cardumes de determinadas espécies de peixe.

O algoritmo começa por gerar posições e velocidades para cada partícula, dentro de uma determinada zona de inicialização. Durante a operação, as velocidades e posições de cada partícula são iterativamente atualizadas até que o critério de finalização seja alcançado.

A cada iteração, cada partícula ajusta a sua direção com base na sua experiência e na experiência da restante população. Cada movimento de cada partícula é baseado essencialmente em três parâmetros: o fator de sociabilidade, o fator cognitivo e a velocidade máxima (Kennedy & Eberhart, 1995).

Cada partícula ajusta a sua velocidade com base no melhor valor que conhece $pBest_i$ (*particle best*) e no melhor valor global descoberto pelas outras partículas $gBest$ (*global best*). Este ajuste tem por base três componentes: a inércia, que previne que a partícula mude de direção abruptamente, agindo como "memória" da posição anterior da partícula; um componente cognitivo, que representa a capacidade de aprendizagem dos padrões de movimento da partícula ($pBest$) e um componente social que quantifica a performance da partícula relativamente a todas as outras ($gBest$) e serve de norma, ou objetivo, a ser atingido por todas as partículas.

Em alguns casos, as partículas podem ser atraídas para regiões fora do espaço delimitado inicialmente, por esta razão, mecanismos para preservar a viabilidade das soluções e a correta operação do enxame foram desenvolvidos (Kennedy & Eberhart, 1995), sendo um dos mecanismos menos disruptivos, de modo a preservar a viabilidade do algoritmo, a limitação à atualização de $pBest_i$ em partículas que estejam próximas de atravessar o espaço delimitado na inicialização do algoritmo, uma vez que ao fazê-lo, a partícula é atraída para áreas que estejam em conformidade com $gBest$ nas iterações seguintes, mantendo-a dentro dos limites.

O algoritmo inicial de PSO proposto por Kennedy e Eberhart (Kennedy & Eberhart, 1995) sofreu algumas alterações ao longo dos anos, dando origem a algumas variantes de PSO, sendo que as principais diferenças das variantes para o algoritmo original são, na sua maioria, relacionadas com a formula que determina a maneira como a velocidade da partícula é calculada.

```
Input: Parâmetros PSO e problema de escalonamento
Ouput: Melhor Solução
Begin
    Inicializar população de partículas
    Avaliar robustez de cada partícula e definir pBest e gBest
    While critério de fim não satisfeito do
        Modificar velocidades com base em pBest anterior e gBest
            Vid = Vi+ri*Ci*(pBesti - Xi) + r2*C2*(gBest-Xi)
        Mover a particula para nova prosição Xi = Xi + Vid
        Avaliar rebustez das particulas individuais
        IF f(Xi) < f(pBesti) then pBesti = Xi
        IF f(Xi) < f(gBest) then gBest = Xi
    End
End
```

Tabela 3 - Algoritmo com base em PSO

3.2.4 Differential Evolution

A *Differential Evolution* (DE) foi introduzida inicialmente por Storn e Price (Storn & Price, 1997) e oferece um método de otimização que não depende da utilização do gradiente do problema para encontrar a sua solução. Isto é particularmente útil se o gradiente é difícil ou impossível de derivar.

O algoritmo de DE mantém uma população de agentes que são iterativamente combinados e atualizados utilizando formulas simples, para criar novas combinações de agentes. A implementação específica do algoritmo deve parametrizar um determinado número de variáveis que influenciam a performance deste comportamento (Storn, 1996).

A DE otimiza o problema mantendo os candidatos com as melhores soluções ao problema durante a iteração do algoritmo como resultado a superar. Desta forma, a otimização do problema é tratada apenas como medida de performance e de qualidade de cada iteração, removendo a necessidade de conhecer o gradiente do problema.

Isto significa que a DE pode ser utilizada em problemas que não sejam contínuos, ou que mudem durante o tempo.

3.2.5 Stochastic Diffusiuon Search

Stochastic Diffusiuon Search (SDS) é um algoritmo de otimização baseado em computação distribuída, onde toda a computação executada pelos agentes é inerentemente probabilística. Os agentes constroem as soluções coletivamente pesquisando de forma independente e difundindo a informação recolhida por toda a população de agentes. O *feedback* positivo promove soluções melhores alocando mais agentes a uma exploração que esteja a ter resultados satisfatórios. A limitação de recursos induz uma forte competição na população, o que indica que soluções com a maior concentração de agentes são estatisticamente as melhores e emergem de forma rápida na execução do algoritmo.

Em muitos problemas de pesquisa, a solução pode ser decomposta em várias sub-regiões, e ao contrário de outros métodos de inteligência enxame, utiliza explicitamente essas decomposições para aumentar a eficiência dos agentes individuais. No SDS, cada agente testa repetidamente uma possível solução para o problema e avalia-a parcialmente. Agentes que testem com sucesso a sua hipótese, rapidamente recrutam outros agentes sem sucesso através de comunicação direta, criando um *feedback* positivo que rapidamente faz convergir os agentes para as soluções promissoras. Regiões da área de soluções onde estejam presentes aglomerados de agentes podem ser interpretadas como candidatos validos para solução do problema. Uma solução global emerge então devido à interação de muitos dos simples agentes que operam localmente (Bishop, 2002).

O algoritmo original de SDS, pesquisa pela melhor solução dado um determinado modelo e uma determinada área de pesquisa, por exemplo, uma palavra em particular (o modelo) num documento de texto (área de pesquisa). Durante o processo de pesquisa, cada agente opera completamente independente de todos os outros, exceto quando entra na fase de troca de informação com a restante população acerca do que encontrou. O SDS não tem um conceito de tempo que leva a um agente a deslocar-se até à posição da hipótese que vai analisar, apenas um contexto espacial (o tamanho da área de pesquisa). Existe, no entanto, um custo associado à avaliação da hipótese, após ser selecionada – o custo da função de avaliação. Durante a execução do SDS, cada agente é capaz de aceder à área de pesquisa completa e de transportar consigo a informação relativa ao modelo alvo, o que no caso da pesquisa de texto, a hipótese a testar pode ser um índice da posição no texto do documento, e esta pode ser avaliada não comparando se a palavra completa (o modelo) se encontra no índice indicado, mas sim verificando apenas um caracter. Caso esta avaliação tenha sucesso, o agente tem a capacidade de comunicar rapidamente com outros agentes (difusão) e estes aglomeram-se nos índices (hipóteses) circundantes para verificar se o restante da palavra (modelo) se encontra também nesses índices até encontrar a totalidade da palavra.

Existem diversas aplicações associadas a SDS que foram aplicadas com sucesso a problemas diversos. Tal como mencionado anteriormente, o SDS foi inicialmente introduzido como um simples algoritmo de pesquisa de texto em 1989 (Bishop, 1989) demonstrando o uso parcial da técnica da função de avaliação, avaliando parcialmente o texto de modo a encontrar o melhor resultado. Subsequentemente, em 1992, uma rede de pesquisa estocástica foi utilizada para localizar posições dos olhos em imagens de faces humanas (Bishop & Torr, 1992). Neste estudo foi demonstrado que o algoritmo conseguia detetar com sucesso as características oculares de todos os sujeitos com que o algoritmo tenha sido treinado, e cerca de 60% em sujeitos desconhecidos. Noutro projeto, em 1995, similar ao anterior, o SDS foi utilizado para detetar expressões faciais (Grech-Cini, 1995) . Em 1998, o SDS foi utilizado para determinar a localização de robôs autónomos num complexo ambiente industrial através de método chamado *Focused Stochastic Diffusion Network* (Beattie & Bishop, 1998). Neste método, o espaço onde os robôs se poderiam encontrar era examinado por um conjunto de agentes que cooperavam entre si de modo a determinar a localização mais provável do robô no ambiente. Em 2002, SDS foi utilizado para determinar a localização da infraestrutura de torres de transmissão de sinais sem fios de modo a manter os custos reduzidos, mantendo a qualidade mínima do serviço e a área de cobertura adequada. Nesta aplicação do algoritmo, dado um conjunto de localizações, este deveria ser capaz de determinar a distribuição das torres de transmissão de modo a que, em qualquer área da rede, fosse possível receber o sinal de pelo menos uma das torres de transmissão (Whitaker, 2002).

3.3 Outras Aplicações

Técnicas baseadas em inteligência enxame podem ser utilizadas num elevado número de aplicações. O exército americano, por exemplo, encontra-se neste momento a investigar técnicas de enxame para controlar veículos não-tripulados, a NASA está a investigar a possibilidade de utilizar tecnologia com base em inteligência enxame para mapear planetas. Em investigação encontram-se problemas como a utilização de robôs microscópicos com base em inteligência enxame para detetar tumores cancerosos (Lewis, 1992), (al-Rifaie, 2012).

A utilização destas técnicas em redes de telecomunicação, cujos pioneiros foram Dorigo e a Hewlett-Packard no início dos anos 90, é também um dos grandes projetos que demonstram a viabilidade destes algoritmos. Com base em ACO, foi possível definir uma tabela de roteamento probabilístico, onde cada formiga (pacote de dados de controlo) procurava preencher toda a rede, reforçando as rotas mais eficazes através da "feromona".

A localização de torres de transmissão, algo essencial na viabilização das infraestruturas de comunicação, é atualmente capaz de ser otimizada através de algoritmos com base em pesquisa de difusão estocástica, uma vez que foi demonstrado que esta heurística aplicada a este caso, consegue providenciar resultados eficazes até mesmo para situações em que o tamanho da rede é extremamente elevado (Whitaker, 2002).

Mais atualmente, o aumento de negócios *data driven*, que sobrevivem e prosperam essencialmente com a base na recolha e análise de elevadas quantidades de informação (*Big Data*) impulsionou o tema de inteligência enxame em contextos académicos e empresariais.

Geralmente, existem duas aproximações ao problema de implementar algoritmos de inteligência enxame a técnicas de *data mining*. A primeira categoria consiste em técnicas onde os indivíduos do enxame movem-se livremente sobre um determinado espaço do problema com objetivo de encontrar soluções para tarefas de análise. Isto é, na aproximação exploratória, os elementos característicos da inteligência enxame são aplicados com objetivo de otimizar as técnicas de *data mining*, como por exemplo melhoramento de parametrizações. Na segunda categoria, é permitido ao enxame movimentar instâncias de dados de caráter pouco dimensional de modo a aglomerá-las em conjuntos sobre os quais é possível realizar análise de dados. Esta é a aproximação organizacional ao problema.

Os algoritmos de inteligência enxame, particularmente ACO e PSO são utilizados em *data mining* para resolver problemas de aglomeração de *clusters* e documentos, parametrização de variáveis, organização de informação e aprendizagem com base em texto, particularmente útil quando os dados têm origem em contextos Web (Pal, 2002).

Como foi possível verificar, existem várias técnicas de otimização com base no paradigma da *swarm intelligence* aplicáveis à resolução de problemas do mundo real. O seu continuo estudo implica quantidades elevadas de experimentação, devido à necessidade de adaptar cada técnica individual à resolução de problemas específicos, no entanto, os possíveis benefícios que podem ser extraídos da correta aplicação destas técnicas à resolução de problemas complexos são demasiado evidentes para que sejam ignorados. O mundo académico caminha claramente no sentido da exploração deste tipo de soluções.

4 Análise experimental

Como componente experimental desta dissertação, foi criado um protótipo de aplicação que tem como objetivo averiguar diferenças entre dois dos algoritmos mencionados no capítulo anterior. A análise de desempenho deve ter em atenção os critérios de otimização que permitam avaliar a eficiência (tempo computacional) e a eficácia (qualidade da solução) dos algoritmos no conjunto de instâncias em análise.

Neste sentido, pretende-se analisar se existe uma diferença significativa no desempenho de técnicas de otimização na resolução de um conjunto de instâncias, do problema de escalonamento com uma só máquina, na minimização dos atrasos pesados *(weighted tardiness*), um dos problemas de escalonamento mencionado no capítulo II.

Pretende-se recorrer a testes paramétricos/não-paramétricos, para realizar a análise de significância dos resultados obtidos, tendo por objetivo concluir quanto à identificação da técnica mais eficiente ou mais eficaz para a resolução da classe de problemas em estudo.

4.1 Arquitetura da solução

A aplicação é composta por três componentes essenciais. O primeiro componente é responsável por parametrizar as restrições do problema e lançar o programa principal com as parametrizações definidas, bem como qualquer dependência.

O segundo componente é efetivamente onde é implementado o problema de SMTWTP a resolver. Este componente tem a capacidade de lançar instâncias do terceiro componente, que

é composto pelas diversas implementações de algoritmos de inteligência enxame que são consideradas neste estudo.

Cada execução do software dá origem a um conjunto de dados, nomeadamente, o tempo de execução e a utilização de memória/CPU, que são guardados numa base de dados para posterior processamento e análise que permite retirar as devidas conclusões.

A arquitetura base da solução pode ser analisada com base no seguinte gráfico (Figura 1):

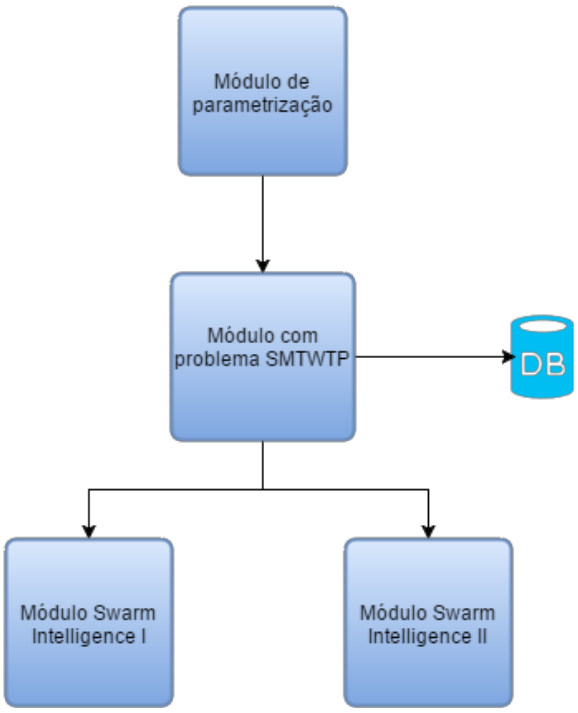

Figura 1 - Arquitetura do prótotipo

A separação dos módulos tem como objetivo extrair as dependências do problema a resolver (SMTWTP, neste caso) das suas parametrizações e da implementação dos algoritmos de inteligência enxame específico. Isto permite realizar testes independentes, com diversas implementações, que possam utilizar código comum.

Deste modo, podemos por exemplo configurar o módulo de parametrização para lançar um número pré-determinado de instâncias do módulo do problema, que por sua vez, por cada instância, lança a instância do algoritmo de inteligência enxame que se pretende testar.

Ao nível da base de dados, a estrutura que sustenta os dados a recolher é muito simples, apenas será registado o qual o algoritmo que correu, o tempo de execução e a qualidade da solução encontrada, de acordo com as tabelas de *SQL* disponíveis no anexo 8.2. Com base nos dados recolhidos é posteriormente feita uma análise estatística.

É possível encontrar no anexo 8.1 um manual de utilizador onde é demonstrado como instalar e executar a aplicação no seu ambiente de desenvolvimento, e como fazer as respetivas configurações necessárias para boa execução (anexo 8.1.2).

4.2 Casos de uso e diagrama de sequência

Para efeitos de interação com o utilizador existirá apenas um caso de uso, que permitirá ao utilizador definir parâmetros e arrancar o teste (Figura 2).

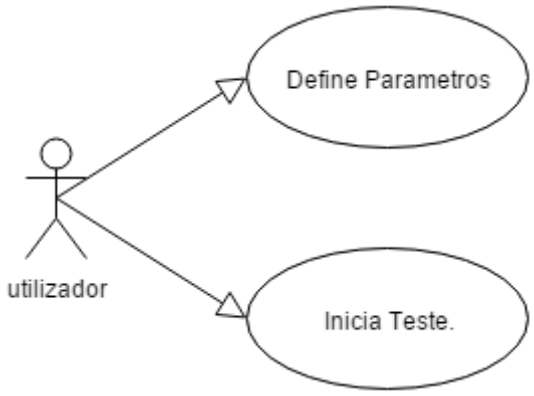

Figura 2 - Caso de uso do utilizador

A aplicação terá a sequência de execução disponível na figura 3. Ao executar a aplicação é executado o módulo responsável pelo carregamento dos parâmetros e instanciação das tarefas a utilizar durante o escalonamento, disponíveis no anexo 8.3. Após carregamento de toda a informação necessária, a aplicação executa o módulo que contém o algoritmo definido na parametrização e atribui-lhe as tarefas carregadas no passo anterior. Os resultados obtidos com a execução deste módulo são devidamente recolhidos e guardados na base de dados após cada execução com sucesso. De notar que cada caso de teste é executado 100 vezes.

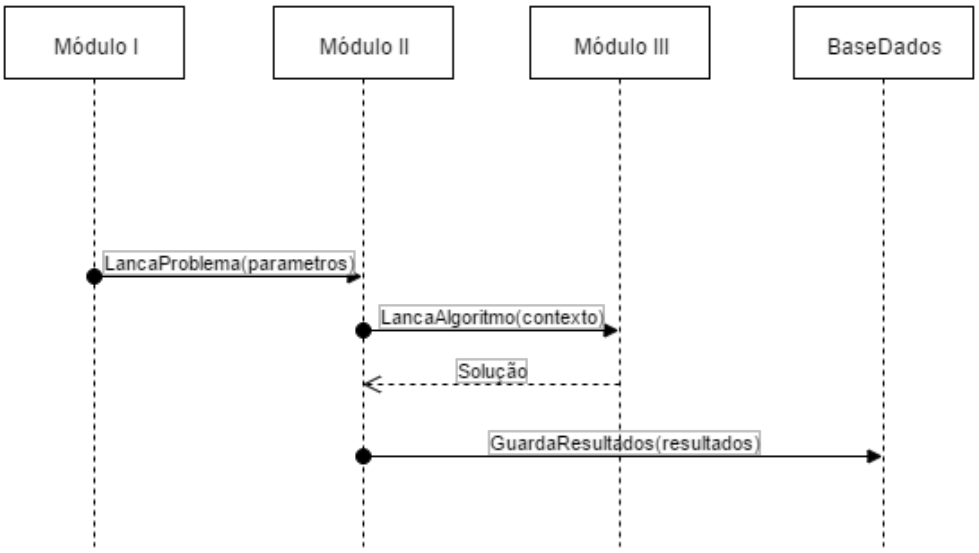

Figura 3 - Diagrama de Sequência

4.3 Algoritmos considerados

Para efeitos de construção da aplicação, foram utilizados como base experimental os algoritmos de *Artificial Bee Colony* (ABC) e *Ant Colony Optimization* (ACO) em sistema Min-Max, ambos descritos no capítulo II desta dissertação.

A aplicação tem como objetivo providenciar soluções para um problema de SMTWTP no menor espaço de tempo possível, tendo em consideração que o resultado pretendido nas execuções de ambos os algoritmos é obter o menor valor de atraso total (*tardiness)* possível.

4.4 Condições de teste

Todos os testes executados no âmbito desta análise experimental foram realizados na mesma máquina, sobre as mesmas condições. Cada ronda de testes foi realizada após uma inicialização completa da máquina, onde o único software que se encontrava em execução no momento de realização do teste, para além do sistema operativo, era, por razões óbvias, o *Visual Studio Community Edition 2015*, visto esta ter sido a ferramenta escolhida para o desenvolvimento do software. A capacidade de processamento do ambiente de teste é dada pelos seguintes componentes:

- **Processador:** Quad-Core Intel i5 2500K @ 4.2GHz (sem Hyper-Threading)
- **RAM**: 8GB DDR3 @ 1866MHz
- **Disco**: Sandisk Ultra II SSD com 128GB de capacidade
- **Sistema Operativo**: Windows 10 Professional, versão de 64 bits
- **Internet**: Fibra a 100Mb
- **Base de dados**: Instância MS SQL Server 2016 em Windows Azure

De notar que apesar dos testes realizados neste âmbito necessitarem de acesso à Internet devido à instância de base de dados estar alojada numa plataforma de *Software as a Service* (SaaS), a mesma pode ser configurada para execução com base de dados alojada localmente, em qualquer outro servidor, desde que este contenha a base de dados *Swarm Intelligence*, definida no anexo 8.2. Em qualquer um dos casos, os problemas inerentes à latência e disponibilidade das comunicações via Internet não têm influência nos resultados dos testes, visto que nenhuma leitura ou inserção de dados é feita durante a execução dos algoritmos a considerar nos mesmos. De notar também que a leitura dos ficheiros que contêm as tarefas a escalonar pelos algoritmos é feita apenas uma vez, no início da execução da solução desenvolvida e colocada em memória até a aplicação terminar, para que não sejam efetuadas leituras de disco adicionais independentemente do número de iterações feitas pelos algoritmos durante cada teste. Deste modo, a leitura do ficheiro no disco não terá influência nos resultados de performance.

A performance foi testada executando vinte e quatro instâncias do problema de *Single Machine Total Weighted Tardiness Problem* (SMTWTP). Estas foram escolhidas com base na sua dimensão (número de tarefas a escalonar) e estão disponíveis na *OR Library* (University of Brunel, 2016), mais concretamente as instâncias de quarenta, cinquenta, e cem tarefas.

Para amenizar influências externas na execução de cada instância, visto não ser possível controlar operações internas do sistema operativo que podem consumir recursos ao sistema durante a execução dos algoritmos, cada instância de teste foi executada cem vezes, com o objetivo de normalizar os resultados obtidos por cada instância do teste.

4.5 Parametrização dos casos de teste

Uma característica em comum partilhada pela maioria dos algoritmos com base em inteligência enxame é o facto de os resultados obtidos dependerem de um conjunto de parâmetros, ajustáveis pelo utilizador e definidos na fase de inicialização dos algoritmos, que contribuem significativamente para a qualidade e/ou performance dos resultados.

De modo a tornar clara a parametrização utilizada na realização dos testes cuja performance será analisada, segue-se uma pequena explicação dos principais parâmetros definidos para cada algoritmo, o porquê da sua utilização, e quais os valores utilizados nas realizações dos testes.

4.5.1.1 Artificial Bee Colony

O algoritmo de *Artificial Bee Colony* tem três parâmetros de elevada importância: o número de origens de alimento, representado no problema de SMTWTP a resolver pelo número de tarefas a escalonar, e de onde é derivado o número de população de abelhas da colmeia, correspondente ao dobro do número de tarefas a escalonar devido à restrição do algoritmo que indica que, na fase de inicialização, o número de abelhas obreiras deve ser igual ao número de abelhas observadoras e que o número de abelhas obreiras é igual ao número de fontes de alimento disponíveis (Karaboga & Akay, 2009). O segundo parâmetro é o número de vezes que uma abelha vai testar uma fonte de alimento até que comece a considerar esta como esgotada. Este limite define o número de vezes que uma abelha testa possíveis localizações de alimento sem sucesso no que diz respeito à quantidade de néctar face às fontes de alimento conhecidas, e que, após atingido, fará a abelha retornar à colmeia para se tornar numa abelha observadora. O terceiro parâmetro é o critério de término do algoritmo, definido pelo número máximo de ciclos a executar.

Para efeitos desta análise experimental, os parâmetros são definidos da seguinte forma:

- **Limite de falha**: 20, aplicável a todas as abelhas artificiais.
- **População de abelhas**: Valores entre 80 e 200, correspondo ao dobro do número de tarefas a escalonar.
- **Critério de término:** Valores correspondentes a 10, 100, 500 e 1000 iterações, dependendo do teste a executar.

4.5.1.2 Ant Colony Optimization

Para o algoritmo de *Ant Colony Optimization* existem vários parâmetros que influenciam a execução do mesmo, sendo os principais o rácio de evaporação da feromona, o número de formigas a gerar, determinado pelo número de tarefas a escalonar no caso do problema de SMTWTP, e o critério de término do algoritmo, dado também pelo número de ciclos a executar.

Dentro do contexto desta análise experimental, o rácio de evaporação da feromona nunca será alterado em nenhum dos casos de teste, de modo a manter a integridade dos mesmos. No entanto, os restantes parâmetros serão alterados em conformidade com as exigências do caso a testar.

Os parâmetros foram definidos da seguinte forma:

- **Rácio de Evaporação**: 10% por iteração.
- **População de formigas**: Valores entre 40 e 100, dependendo do número de tarefas a escalonar.
- **Critério de término:** Valores correspondentes a 10, 100, 500 e 1000 iterações, dependendo do teste a executar.

4.5.2 Casos de teste

Tal como referido anteriormente, as instâncias do problema SMTWTP a considerar nos testes são obtidas através da *OR Library,* mais especificamente as instâncias disponíveis nos ficheiros wt40.txt, wt50.txt e wt100.txt. Estes ficheiros possuem instâncias do problema com quarenta, cinquenta e cem tarefas a escalonar, respetivamente. As tarefas utilizadas em todos os testes estão disponíveis nos anexos 8.3.1, 8.3.2 e 8.3.3 pela mesma ordem, de acordo com a quantidade de tarefas a escalonar.

ALGORITMO	NÚMERO TAREFAS	ITERAÇÕES
ACO	40	10
ACO	40	100
ACO	40	500
ACO	40	1000
ACO	50	10
ACO	50	100
ACO	50	500
ACO	50	1000
ACO	100	10
ACO	100	100
ACO	100	500
ACO	100	1000
ABC	40	10
ABC	40	100
ABC	40	500
ABC	40	1000
ABC	50	10
ABC	50	100
ABC	50	500
ABC	50	1000
ABC	100	10
ABC	100	100
ABC	100	500
ABC	100	1000

Tabela 4 – Casos de teste

Para cada conjunto de tarefas foram realizados quatro testes, diferenciados pela parametrização da sua execução, com especial atenção ao número de iterações do algoritmo, de acordo com a tabela 4.

Como referido anteriormente, cada teste é executado cem vezes, com o objetivo de normalizar os resultados, dando origem a uma base dados com dois mil e quatrocentos registos, que serão sujeitos a análise.

Um dos objetivos da experimentação com base nestas parametrizações é não só medir e comparar tempos de execução *versus* a qualidade da solução obtida, mas também determinar se, e em que medida, o número de iterações influencia a qualidade dos resultados obtidos.

Da execução destes testes pretende-se guardar alguns indicadores de performance como o tempo de execução e o melhor resultado conseguido pelo algoritmo.

Após a recolha dos dados de execução para os casos de teste especificados, é feita uma análise individual de cada algoritmo para cada grupo de tarefas a escalonar, de onde se retiram conclusões face à influência do número de iterações do algoritmo em análise no desempenho do escalonamento das tarefas. Finalmente, é feita também uma análise comparativa entre os dois algoritmos na resolução do problema de SMTWTP para os casos em teste.

4.6 Detalhes da implementação

4.6.1 Tecnologias utilizadas

O protótipo utilizado foi desenvolvido com base em diversas tecnologias *Microsoft*, nomeadamente:

- **Linguagem**: C Sharp (C#), desenvolvida pela *Microsoft* como parte de iniciativa *.NET*.
- **Ambiente de Desenvolvimento**: *Microsoft Visual Studio* 2015 *Community Edition,*
- **Base de dados**: *Microsoft SQL Server*

4.6.2 Heurística de determinação da qualidade da solução

É possível argumentar que um dos fatores mais importantes para a performance dos algoritmos é a pesquisa local realizada por cada agente individual a cada iteração. Em

situações normais é aplicada alguma heurística pré-definida de acordo com o objetivo do problema a resolver. Uma vez que o problema em estudo é o SMTWTP, o objetivo é determinar o caminho com o menor atraso/tardeza possível. Para tal, em ambos os algoritmos é utilizada uma medida de performance para determinar a qualidade da solução encontrada de acordo com a seguinte formula:

$$Cost_s = TaskWeight_i * Max(0, Time - TaskDeadline_I)$$

Em termos de código isto reflete-se no algoritmo de ABC na função que determina o *fitness* da solução encontrada pela abelha (Figura 4). A função recebe como parâmetro a lista ordenada pela abelha, que no contexto do problema corresponde a uma solução encontrada, e testa a sua qualidade de acordo com a formula referida anteriormente, calculando a diferença entre o tempo corrido e a *deadline* da tarefa, se o resultado desta diferença for um valor positivo, significa que a tarefa na ordenação em que se encontra irá gerar atraso, e por isso o valor resultante é multiplicado pelo valor do peso da tarefa. Este processo é repetido por todas as tarefas e no final a soma total do atraso corresponde ao valor pelo qual a qualidade do resultado é averiguado, sendo o valor mais baixo correspondente à melhor solução.

```
private static double GetSolutionFitness(List<WtTask> tasksToSchedule)
        {
            var argClone = Clone(tasksToSchedule);

            var elapsedTime = 0;
            var cost = 0;

            foreach (var task in argClone)
            {
                elapsedTime += task.ExecutionTime;
                cost += task.Weight * Math.Max(0, elapsedTime - task.Deadline);
            }

            return Convert.ToDouble(cost);
        }
```

Figura 4 - *GetSolutionFitness()* em ABC

No caso do algoritmo de ACO, a função é semelhante (Figura 5). Enquanto não forem processadas todas as tarefas ordenadas pela formiga é calculado o peso da tarefa atual utilizando a mesmo formula que no processo anterior, e o seu custo adicionado a um valor total, que corresponde ao custo da solução.

```
internal void Walk()
{
        _mCost = Time = 0;
        while (_freeTasks.Count != 0)
        {
            var task = _mChooseWay();
            Path.Add(task);
            Time += _mTaskses.MExecutionTime[task];
            _mCost += _mTaskses.MWheight[task] * Math.Max(0, Time - Tasks.MDea
dlines[task]);
        }
}
```

Figura 5 - *SolutionCost* em ACO

5 Análise e discussão dos resultados

Este capítulo serve de discussão sobre os resultados obtidos através da execução de todos os casos de teste mencionados no capítulo anterior. Pretende-se fazer uma análise resumida e individual, dos resultados de cada grupo de testes realizados e discutir algumas conclusões retiradas da análise dos dados recolhidos.

5.1 ACO com escalonamento de quarenta tarefas

O primeiro caso em análise consiste no escalonamento de quarenta tarefas, tendo o algoritmo executado com diferentes valores de iterações, entre as dez e as mil, de acordo com o a tabela 5.

Para a execução com dez iterações, obtivemos o valor mínimo de quarenta e oito milissegundos de tempo de execução e um custo de tardeza total de mil quatrocentos e cinquenta e um. Os valores máximos correspondem a um tempo de oitocentos e cinquenta e nove milissegundos de tempo de execução e quatro mil setecentos e vinte e três de custo da solução. Os valores médios para as dez iterações situam-se nos setecentos e noventa e três milissegundos de tempo de execução e três mil trezentos e cinquenta e sete de tardeza de solução. De notar que a diferença entre o valor máximo e o valor mínimo de tardeza para este caso de teste corresponde a um aumento de duzentos e vinte cinco por cento da qualidade da melhor solução encontrada.

Na execução com cem iterações podemos observar que o valor de tardeza mínimo obtido foi superior ao obtido no teste anterior, no entanto, o valor máximo e a média

correspondem a valores melhores, continuando a sugerir que o aumento do número de iterações corresponde à obtenção de melhores valores gerais.

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ACO	40	10	Tempo (ms)	100	48.0	859.0	793.190	169.7836
			Tardeza Solução	100	1451.0	4723.0	3357.480	678.0657
		100	Tempo (ms)	100	469.0	1293.0	843.230	103.6400
			Tardeza Solução	100	1861.0	3966.0	3036.860	422.3841
		500	Tempo (ms)	100	2155.0	2820.0	2237.330	88.6334
			Tardeza Solução	100	1734.0	3131.0	2625.840	346.6961
		1000	Tempo (ms)	100	4299.0	6444.0	4660.210	212.8455
			Tardeza Solução	100	1841.0	3348.0	2611.160	288.0492

Tabela 5 - Estatísticas descritivas para 40 tarefas

Para o teste de quinhentas iterações podemos verificar que essa tendência continua, os tempos médio de tardeza da solução situam-se nos dois mil seiscentos e vinte cinto, uma melhoria de quinze por cento face ao teste anterior.

No teste de mil iterações, os valores médios tendem novamente a melhorar, no entanto, é de notar que a diferença entre a execução com quinhentas e mil iterações produz resultados médios com um diferencial inferior a 1%, mas onde a execução com mil iterações demora mais do dobro do tempo, de acordo com a figura 7

É possível observar através da análise do histograma (Figura 6) que o aumento do número de iterações faz com que os resultados convirjam para um conjunto de resultados tendencialmente mais próximos da média e com distribuição inferior, facto aliás suportado pela redução do desvio padrão em cada caso de teste, à medida que o numero de iterações aumenta.

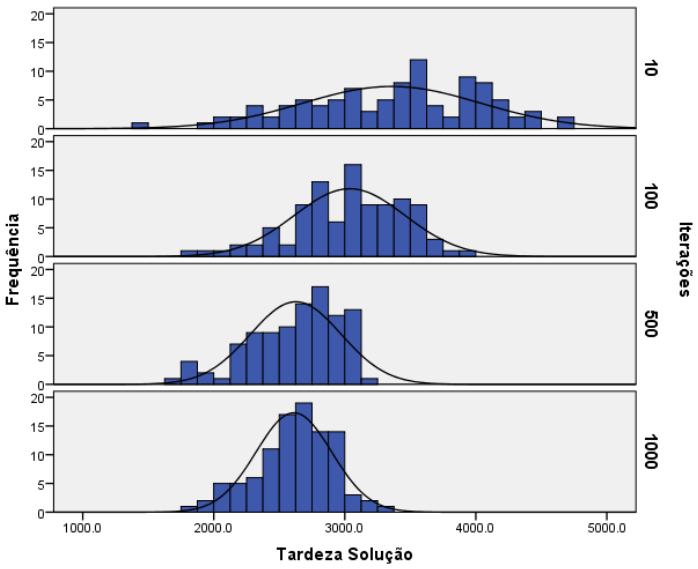

Figura 6 - Qualidade da solução para 40 tarefas

No que diz respeito aos tempos de execução, podemos verificar na tabela 7 que apesar de a diferença entre as execuções com dez e cem iterações não ser muito relevante, situando-se aproximadamente nos novecentos milissegundos de tempo médio, o aumento do tempo de execução dispara nas iterações subsequentes de forma notória, correspondendo a uma diferença de cento e sessenta e cinco por cento nos tempos médios de execução entre as cem e as quinhentas iterações, e aproximadamente cento e oito por centro entre as quinhentas e as mil iterações, sendo que neste último caso, o tempo de execução se situa por volta dos quatro segundos e meio.

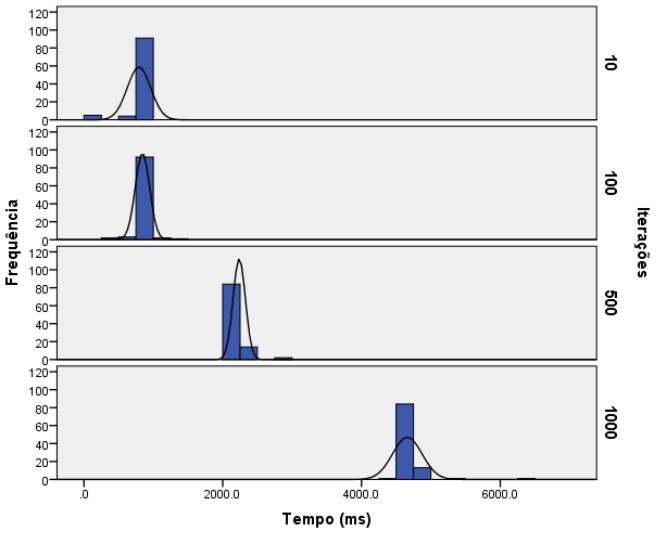

Figura 7- Tempos de execução para 40 tarefas

Podemos ainda observar que o algoritmo de ACO foi capaz de gerar a melhor solução do grupo correndo apenas com dez iterações, no entanto, os testes executados com dez iterações são também os que obtiveram a pior média de resultados e o maior desvio padrão do grupo. As execuções com mil iterações oferecem os melhores resultados absolutos, mas também os piores tempos de execução, sendo a melhoria na qualidade do resultado diminuta face aos resultados obtidos com quinhentas iterações.

5.2 ACO com escalonamento de cinquenta tarefas

No escalonamento de cinquenta tarefas com ACO temos um cenário relativamente similar ao anterior, o que é expectável devido ao aumento de apenas 20% das tarefas a escalonar, como pode ser visto na tabela 6. Para o teste com dez iterações obtiveram-se valores de tardeza média perto dos quatro mil e quinhentos, mas com tempos de execução com apenas mais dois milissegundos que o caso de teste anterior. Fenómeno normal, visto que o aumento da tardeza é espectável com o aumento do número de tarefas. Para o teste de cem iterações a tardeza média melhorou cerca de doze por cento face ao teste anterior, no entanto, o tempo de execução quase dobrou. Para o teste de quinhentas iterações obteve-se uma melhoria de dez por cento face ao resultado anterior, mas novamente, o tempo de execução quase triplicou, começando a ser óbvio o menor retorno na qualidade da solução, face ao tempo de execução necessário para o obter.

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ACO	50.0	10	Tempo (ms)	100	76.0	1351.0	795.810	174.1714
			Tardeza solução	100	2887.0	5668.0	4469.510	575.8495
		100	Tempo (ms)	100	783.0	1819.0	1293.830	151.8475
			Tardeza solução	100	2818.0	4741.0	3914.290	403.9867
		500	Tempo (ms)	100	3822.0	4837.0	3962.720	109.3584
			Tardeza solução	100	2432.0	4221.0	3493.800	332.9703
		1000	Tempo (ms)	100	7222.0	9218.0	7752.540	295.4678
			Tardeza solução	100	2581.0	3940.0	3307.120	336.8938

Tabela 6 - Estatísticas descritivas para 50 tarefas

Nada ilustra melhor o pior retorno na qualidade dos resultados face ao tempo de execução do que o teste de mil iterações. Quando comparado com o de quinhentas, obteve apenas uma melhoria média de cinco por cento na qualidade da solução, mas demorando quase oito segundos, ou seja, o dobro, novamente, para obter essa melhoria. No entanto, podemos verificar que há uma progressão clara dos valores médios gerados pelo algoritmo para a qualidade da solução, consoante o número de iterações (Figura 8). Mais uma vez, é óbvio que com o aumento do número de iterações, a distribuição dos resultados tende a convergir para valores próximos da média.

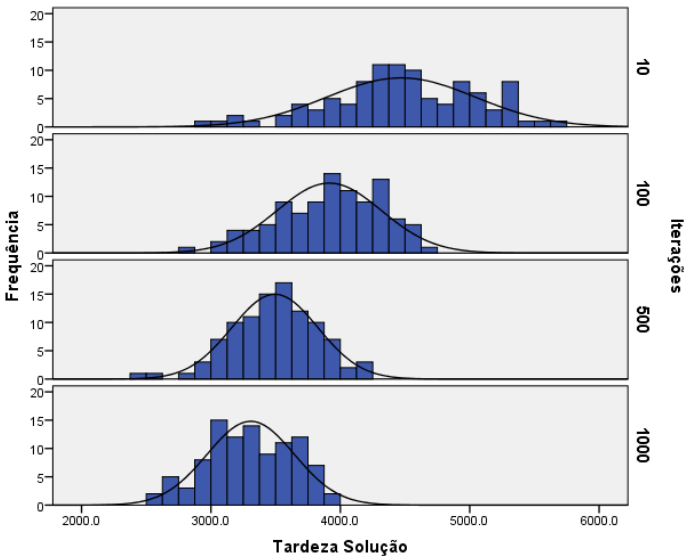

Figura 8 - Qualidade da solução para 50 tarefas

Respetivamente aos tempos de execução, é notável que com o aumento de iterações em cada teste, os valores tendem a duplicar ou até triplicar em comparação com o número de iterações anterior (Figura 9), tornando-se claro que a melhoria da qualidade pode não justificar necessariamente o tempo de execução necessário para a obter.

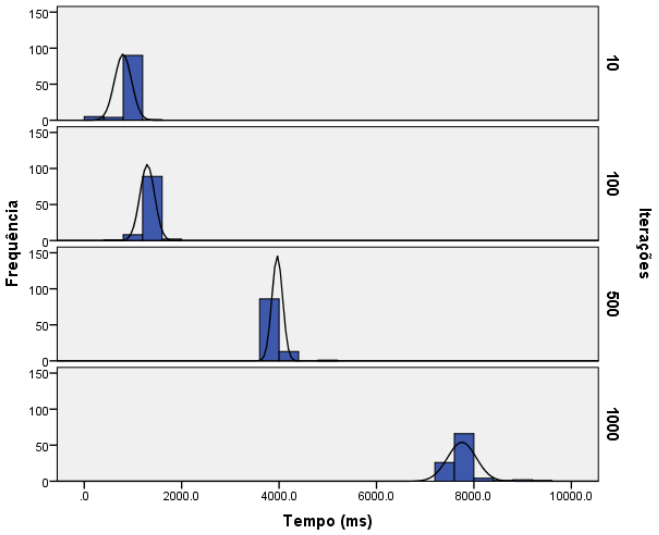

Figura 9 - Tempos de execução para 50 tarefas

5.3 ACO com escalonamento de 100 tarefas

Para cem tarefas, é possível observar um claro aumento dos tempos de execução em função do número de iterações, no entanto, podemos também observar que em termos médios, a qualidade das soluções apresenta pouca variação, sugerindo que para este teste o número de iterações do algoritmo terá pouca influência na qualidade do resultado (Tabela 7).

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ACO	100.0	10	Tempo (ms)	100	422.0	1249.0	815.810	95.7951
			Tardeza solução	100	14061.0	25599.0	21245.870	2282.8617
		100	Tempo (ms)	100	4060.0	4551.0	4368.060	90.3596
			Tardeza solução	100	16571.0	24852.0	20888.940	1844.4009
		500	Tempo (ms)	100	20145.0	24445.0	21223.120	813.2232
			Tardeza solução	100	15503.0	24620.0	20555.530	1895.6787
		1000	Tempo (ms)	100	40329.0	49628.0	42304.270	1226.5419
			Tardeza solução	100	14411.0	25098.0	20439.290	2321.9008

Tabela 7 - Estatística descritiva para 100 tarefas

Podemos observar que a tardeza média aumentou significativamente face aos testes de cinquenta tarefas, no entanto, esse facto é relativamente esperado, visto que ao escalonar tarefas onde não existe a solução ótima (anexo 8.3.3), correspondente a uma tardeza total de 0, quanto mais tarefas for necessário escalonar, maior a tardeza total esperada.

No teste de dez iterações, obtivemos valores médios de tardeza de vinte e um mil duzentos e quarenta e cinco, num tempo de execução também médio de oitocentos e quinze milissegundos.

Nos testes com número de iterações superiores, podemos observar a abismal performance do algoritmo no que diz respeito a tempos de execução. A diferença entre o valor médio de tardeza com dez iterações e o valor médio de tardeza com mil iterações é de apenas quatro por cento, no entanto, demora cerca de cinquenta e duas vezes mais em execução para obter essa melhoria. Inclusive, analisando os resultados mínimos é possível verificar que a execução com dez iterações obteve também o melhor resultado geral. Isto

aparenta sugerir que à medida que o número de tarefas a escalonar aumenta, o número de iterações torna-se num fator menos relevante na qualidade do resultado.

Podemos também observar que em termos da distribuição dos resultados (Figura 10), a curva é significativamente mais dispersa do que nos casos anteriores, salientando a fraca relação entre o número de iterações e a qualidade do resultado para este teste.

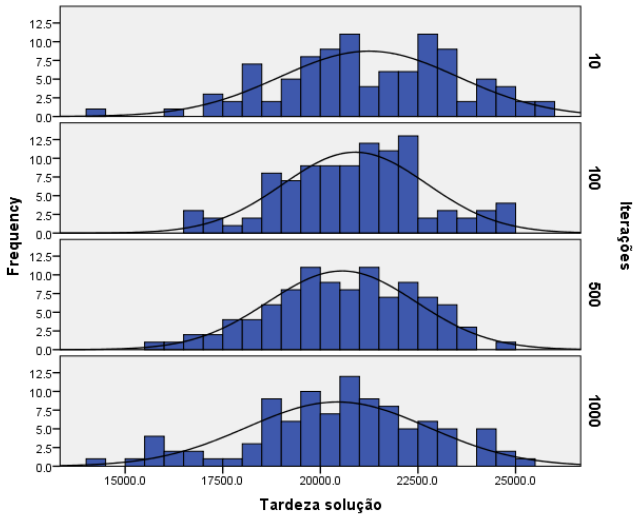

Figura 10 - Qualidade da solução para 100 tarefas

A distribuição dos tempos de execução (Figura 11), representa sim, uma clara relação entre o numero de iterações e o tempo de execução necessário para obter os resultados. Tal como referido anteriormente, com o aumento de iterações a performance do algoritmo demonstra uma clara deterioração nos tempos de execução, chegando a demorar cinquenta e duas vezes mais quando medidos os extremos para obter uma melhoria de quatro por cento. De salientar que a execução do teste com cem iterações demorou quase uma hora a concluir.

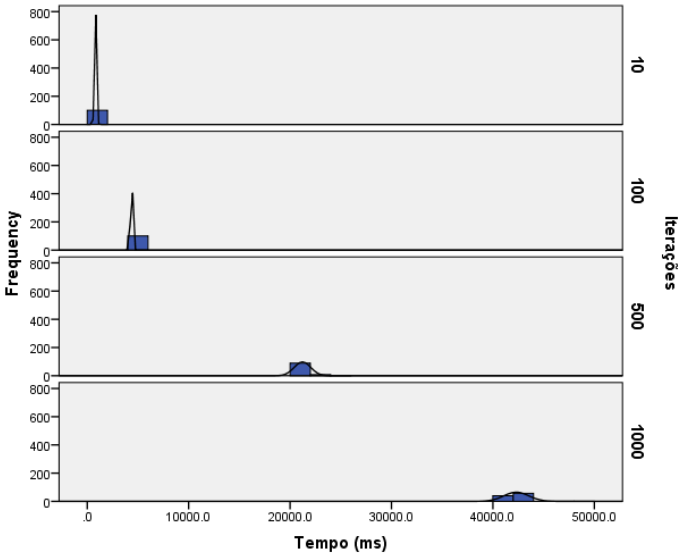

Figura 11 - Tempos de execução para 100 tarefas

5.4 ABC com escalonamento de 40 tarefas

Alterando o algoritmo e regressando ao teste de quarenta tarefas, é possível observar claramente duas diferenças, primeiramente, os tempos de execução para quarenta tarefas são largamente inferiores aos obtidos no algoritmo de ACO, em segundo lugar, a qualidade da solução obtida é igualmente inferior.

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ABC	40.0	10	Tempo (ms)	100	8.0	18.0	9.800	1.2472
			Tardeza Solução	100	26772.0	35943.0	31853.840	1850.6305
		100	Tempo (ms)	100	69.0	105.0	82.630	7.0691
			Tardeza Solução	100	26451.0	33337.0	29913.790	1597.6312
		500	Tempo (ms)	100	409.0	491.0	446.500	18.2433
			Tardeza Solução	100	24322.0	29693.0	27132.580	1134.4635
		1000	Tempo (ms)	100	874.0	1381.0	955.440	67.7593
			Tardeza Solução	100	24114.0	28243.0	25923.440	933.5420

Tabela 8 - Estatística descritiva para 40 tarefas

Para o teste de dez iterações, podemos verificar que a tardeza de solução é claramente superior às obtidas no mesmo teste com o algoritmo de ACO, no entanto o tempo de execução necessário é bastante inferior, situando-se em média nos dez milissegundos. Aumentando as iterações para cem podemos verificar que a melhoria na qualidade de solução aumenta apenas cerca de seis por cento, em média. Já o tempo de execução quase multiplica por dez. No entanto, continua a manter valores inferiores aos obtidos no algoritmo de ACO.

Subindo as iterações para quinhentas obtemos uma melhoria na solução na ordem dos nove por centro face ao teste anterior, mas o tempo de execução quadruplica também, mantendo-se mesmo assim abaixo de meio segundo.

No último caso de teste para este número de tarefas, obtemos uma melhoria média de cinco por cento, sendo o custo de tempo o dobro do teste anterior. De salientar que mesmo no teste com mil iterações, os tempos de execução foram significativamente mais baixos face aos obtidos com o algoritmo de ACO para o escalonamento de quarenta tarefas.

Quanto à distribuição dos resultados, podemos observar que com o aumento do número de iterações os resultados tendencialmente ficam ligeiramente mais consistentes (Figura 12), mas mesmo assim de modo não tão aparente face ao obtido com o ACO

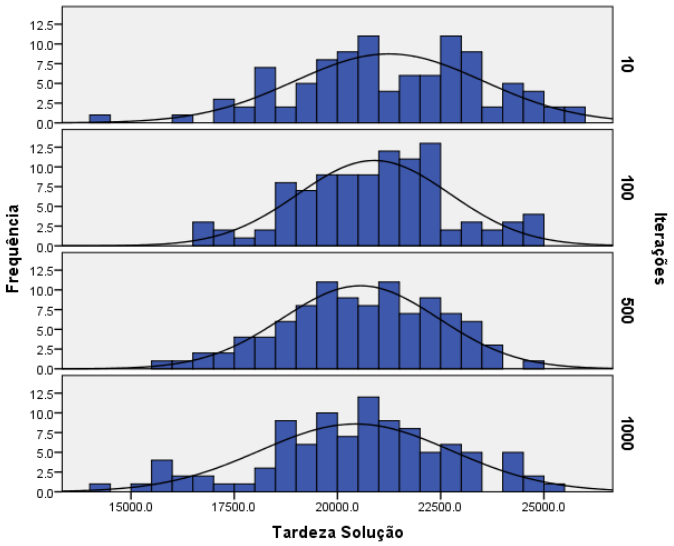

Figura 12 - Qualidade da solução para 40 tarefas

Relativamente ao tempo de execução, podemos observar que, tal como referido anteriormente, estes são largamente menores do que todas as execuções até ao momento (Figura 13), ficando as execuções de mil iterações, bastante perto de um segundo de execução, valor claramente inferior face ao obtido nas mesmas condições com ACO. O aumento de tempo depende claramente do número de execuções no escalonamento de quarenta tarefas, não existindo nenhum caso em que o tempo de execução não dobre, pelo menos, quando aumentado o número de iterações.

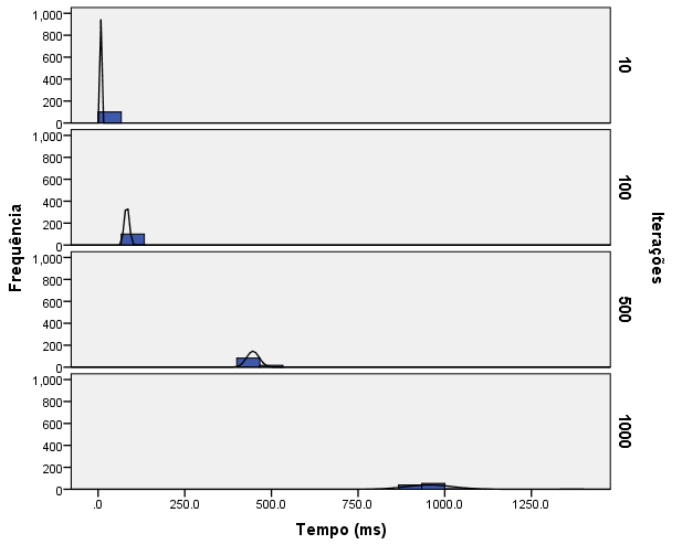

Figura 13 - Tempos de execução para 40 tarefas

5.5 ABC com escalonamento de 50 tarefas

Para o algoritmo de ABC com cinquenta tarefas podemos verificar que há o aumento esperado quer na tardeza da solução, bem como no tempo de execução (Tabela 9).

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ABC	50.0	10	Tempo (ms)	100	13.0	25.0	16.030	1.5920
			Tardeza Solução	100	36189.0	47631.0	42556.910	2288.6826
		100	Tempo (ms)	100	108.0	214.0	132.630	17.2572
			Tardeza Solução	100	35597.0	43551.0	39778.400	1904.1598
		500	Tempo (ms)	100	657.0	1138.0	743.790	74.3893
			Tardeza Solução	100	32271.0	38409.0	35699.290	1269.2379
		1000	Tempo (ms)	100	1382.0	1674.0	1517.180	54.9478
			Tardeza Solução	100	31592.0	37630.0	34115.240	1205.0632

Tabela 9 - Estatística descritiva para 50 tarefas

No entanto, podemos também concluir que a tardeza média entre os quatro grupos de iterações tem uma diferença máxima de vinte e dois por cento, mas a diferença de tempo é muito mais significativa, indicando assim que o simples aumento do número de iterações não tem um retorno na qualidade igualitário. O teste de dez iterações apresenta uma tardeza de solução bastante superior quando comparada com o mesmo teste com quarenta tarefas, o que embora um aumento fosse esperado, não seria necessariamente tão grande. Já o tempo de execução médio passa para dezasseis milissegundos, dobrando assim o valor do teste anterior. O teste de cem iterações apresenta uma melhoria de cerca de seis por cento e um aumento de tempo de execução de quase setecentos por cento, situando-se agora nos cento e trinta e dois milissegundos. Aumentando para quinhentas iterações a melhoria de tardeza média sobe cerca de dez por cento, um aumento considerável, e o tempo de execução quadruplica, mantendo-se mesmo assim bastante abaixo do obtido no mesmo teste com o algoritmo de ACO. No teste com mil iterações temos uma melhoria de apenas quatro por cento na tardeza média, mas dobrando o tempo de execução, que sobe agora acima de um segundo, valor este bastante inferior ao obtido no algoritmo de ACO no mesmo teste.

Quanto à distribuição dos resultados, observamos o mesmo padrão demonstrado até esta fase, o aumento das iterações tende a resultar numa melhor normalização dos resultados, bem como melhor qualidade dos resultados no geral (Figura 14).

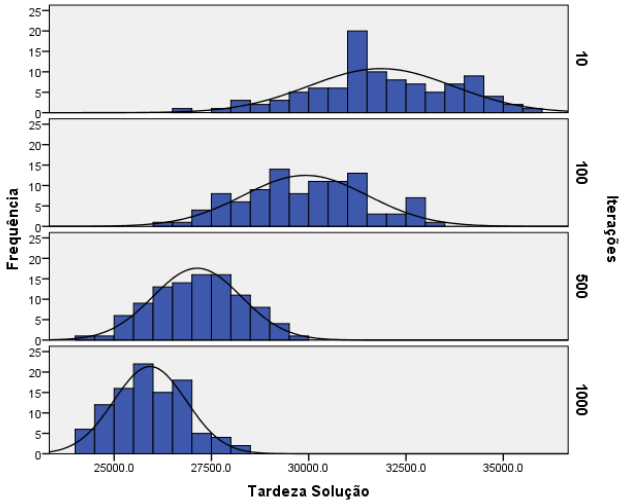

Figura 14 - Qualidade da solução em ABC para 50 tarefas

No entanto, e tal como referido anteriormente, a partir das quinhentas iterações a melhoria relativamente pequena na qualidade do resultado é obtida com um maior gasto de tempo, obtendo assim, retornos mais pequenos apesar do maior gasto temporal (Figura 15).

A distribuição dos tempos de execução varia largamente conforme o número de iterações, entre dezasseis milissegundos e um segundo e meio.

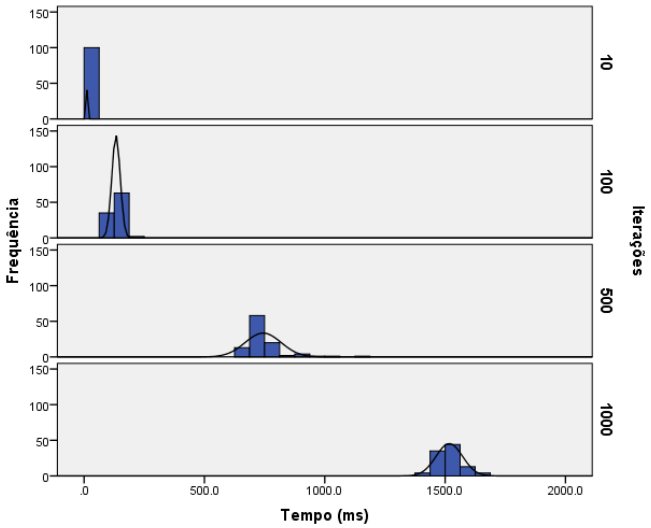

Figura 15 - Tempos de execução para ABC a 50 tarefas

5.6 ABC com escalonamento de 100 tarefas

O último caso em análise, ABC com escalonamento de cem tarefas, apresenta resultados bastante pobres relativamente à tardeza de solução média, existindo um diferencial de aproximadamente dezasseis por cento entre os valores médios de tardeza contemplados entre todas as iterações (Tabela 10).

Algoritmo	Tarefas	Iterações		N	Mínimo	Máximo	Média	Desvio Padrão
ABC	100.0	10	Tempo (ms)	100	76.0	112.0	89.380	5.5465
			Tardeza Solução	100	182570.0	217554.0	199159.140	6231.9676
		100	Tempo (ms)	100	252.0	804.0	564.520	90.3598
			Tardeza Solução	100	172394.0	198867.0	185452.650	5050.5917
		500	Tempo (ms)	100	3135.0	4379.0	3764.280	241.9952
			Tardeza Solução	100	162369.0	180586.0	173022.270	3407.0033
		1000	Tempo (ms)	100	7442.0	9105.0	8256.830	349.6704
			Tardeza Solução	100	158903.0	178507.0	169179.840	3327.9273

Tabela 10 - Estatística descritiva para ACB a 100 tarefas

Verificamos no teste com dez iterações que para o escalonamento de cem tarefas o valor médio de tardeza obtido sobe consideravelmente, para valores médios quase dez vezes superiores aos obtidos no algoritmo de ACO para o mesmo caso de teste. O aumento para cem iterações representa uma melhoria de sete por cento na tardeza média obtida, subindo também o tempo de execução acima de quinhentos milissegundos. No teste de quinhentas iterações a tardeza média melhorou novamente sete por cento, mas o tempo de execução subiu consideravelmente, para valores próximos dos quatro segundos. Uma melhoria nada substancial. No entanto, no caso de teste de mil iterações vimos fraquíssimo retorno na qualidade, visto que a diferença de valores de tardeza melhora apenas dois por cento, enquanto que o tempo necessário para obter essa melhoria mais do que duplica, subindo o tempo de execução acima dos oito segundos, em média. É, portanto, a pior melhoria em termos gerais

nos testes realizados até ao momento. É possível observar na figura 16 que, tal como nos testes realizados anteriormente a distribuição dos resultados demonstra tendência e melhorar com o aumento do número de iterações realizado.

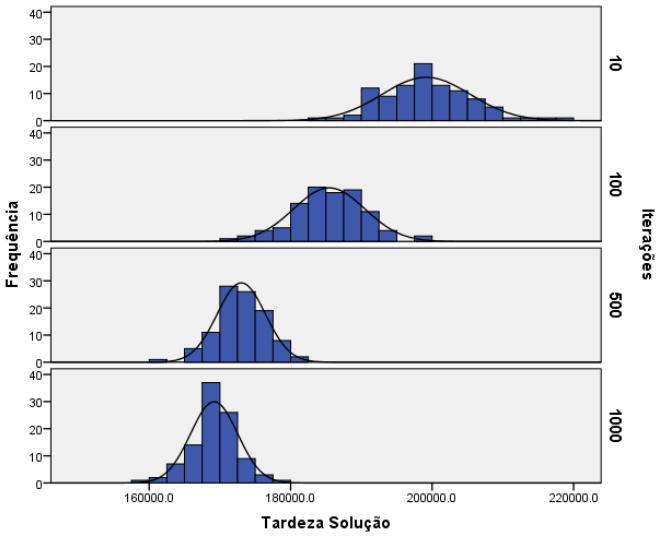

Figura 16 - Qualidade da solução para ABC a 100 tarefas

Relativamente aos tempos de execução observados, podemos verificar através da figura 17 que a distribuição dos tempos de execução é relativamente consistente e que, mesmo no escalonamento de cem tarefas com mil iterações, situação que no algoritmo de ACO necessitava de mais de quarenta segundos para executar, o ABC necessita apenas de oito, tornando este algoritmo bastante interessante quando for necessário obter soluções rapidamente.

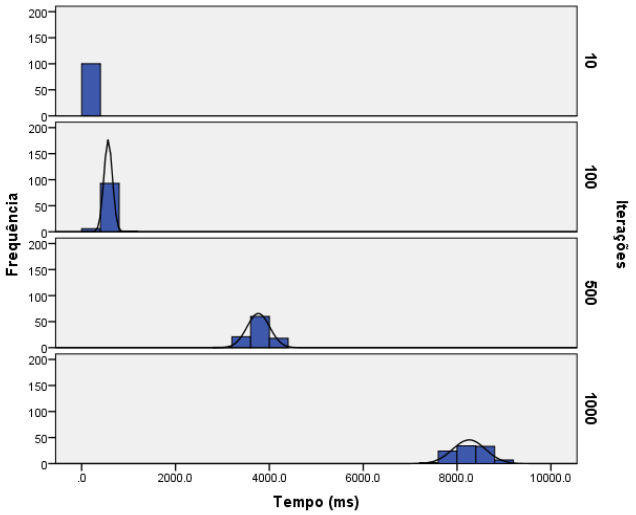

Figura 17 - Tempos de execução para ABC a 100 tarefas

5.7 Observações sobre os resultados

A conclusão da fase de testes e análise dos seus resultados leva a que seja possível retirar algumas conclusões, uma vez que os dados recolhidos indicam algumas diferenças inerentes no comportamento dos algoritmos em análise devido às suas características individuais.

A primeira conclusão, e possivelmente a mais óbvia, é que, em todos os casos de teste, é visível a existência de uma correlação entre o número de iterações do algoritmo testado, e a qualidade do seu resultado final e/ou tempo de execução, o que leva a que se coloque a hipótese de existência de uma influência do número de iterações na qualidade dos resultados obtidos.

Para validar esta hipótese foi realizado um teste estatístico de regressão linear para analisar a relação entre a variável dependente (tardeza da solução) e a variável independente (número de iterações) em ambos os algoritmos, para o exemplo de escalonamento de cem tarefas.

Para o algoritmo de ACO, uma equação de regressão estatisticamente significante foi encontrada, $(f(1, 398) = 7.236, p < 0.007)$, com um R^2 de 0.018. A tardeza prevista do algoritmo é igual a 21072.59 -7.21 (iterações), ou seja, a tardeza do algoritmo diminui 7.21 unidades por cada iteração adicional.

Analisando a distribuição de probabilidade (Figura 18), podemos observar visualmente a existência da correlação esperada entre as variáveis.

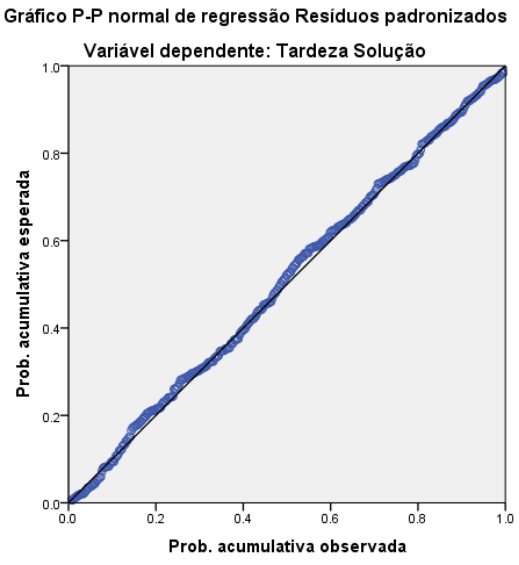

Figura 18 - P-P Plot da regressão do algoritmo ACO

Para o algoritmo de ACO, o mesmo teste de regressão foi efetuado, contemplando também o caso de escalonamento de cem tarefas. Uma equação estatisticamente significante foi encontrada, $(f(1,398) = 855{,}750, p < 0.0)$, com um R^2 de 0.683. A tardeza esperada do algoritmo é de 192432.1 – 26,655 (iterações), ou seja, a tardeza do algoritmo diminui 26,655 unidades por cada iteração adicional. Analisando a distribuição de probabilidade (Figura 19), podemos observar visualmente a existência da correlação esperada entre as variáveis.

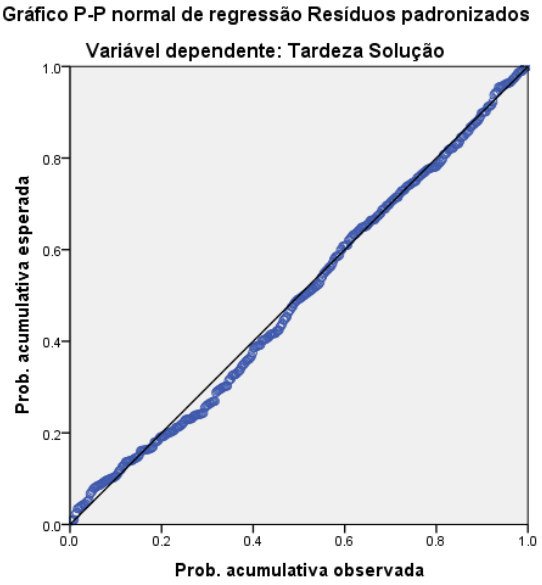

Figura 19 P-P Plot da regressão do algoritmo ABC

A realização dos testes de regressão permite não só estabelecer a relação direta entre o número de iterações e a qualidade do resultado obtido, mas permite também explicar algumas das características individuais da execução de cada um dos algoritmos. Por exemplo, no caso do algoritmo de ACO, foi possível verificar através do valor de R^2, que o número de iterações explica apenas 1,8% da variância da tardeza da solução, enquanto que no algoritmo de ABC, o número de iterações é responsável por 68% da variância. A pouca influência do número de iteração nos testes realizados com o algoritmo de ACO é explicada devido aos efeitos da feromona na pesquisa local efetuada por cada formiga. Esta torna a complexidade do algoritmo mais elevada, mas, presumindo que a formiga não opta por um caminho aleatório, e que a feromona não evapora entretanto, tem a vantagem de permitir que o melhor caminho para a solução esteja disponível para todas as formigas ativas nas iterações subsequentes, chegando assim a um resultado aceitável com um número de iterações reduzido.

Já no algoritmo de ABC, o comportamento não é similar, o foco é encontrar possíveis soluções para o problema o mais rápido possível e explorar soluções vizinhas com a ajuda do

resto da colónia, facto que se reflete na rapidez de execução do algoritmo. Como última observação dos resultados, visto que o objetivo desta análise experimental é determinar qual o algoritmo mais adequado para a resolução de problemas de SMTWTP, é possível verificar que em todos os casos de teste analisados anteriormente, o algoritmo de ACO obteve uma performance claramente superior se contemplarmos que o objetivo principal é obter a menor tardeza possível, num tempo de execução relativamente reduzido (Tabela 11).

Algoritmo	Tarefas		Mínimo	Máximo	Média	Desvio padrão
		Tempo (ms)	8.0	1381.0	373.593	376.5668
	40.0	Tardeza Solução	24114.0	35943.0	28705.913	2725.5568
		Tempo (ms)	13.0	1674.0	602.408	598.6811
ABC	50.0	Tardeza Solução	31592.0	47631.0	38037.460	3750.1851
		Tempo (ms)	76.0	9105.0	3168.752	3271.1576
	100.0	Tardeza Solução	158903.0	217554.0	181703.475	12636.4204
		Tempo (ms)	48.0	6444.0	2133.490	1579.0031
	40.0	Tardeza Solução	1451.0	4723.0	2907.835	552.8019
		Tempo (ms)	76.0	9218.0	3451.225	2770.2161
ACO	50.0	Tardeza Solução	2432.0	5668.0	3796.180	615.1633
		Tempo (ms)	422.0	49628.0	17177.815	16464.9663
	100.0	Tardeza Solução	14061.0	25599.0	20782.407	2113.1799

Tabela 11 - Resultados gerais de ACO e ABC

Podemos observar que, mesmo ignorando o número de iterações e olhando apenas para os valores absolutos dos testes, em nenhum dos escalonamentos, independentemente do número de tarefas ou iterações, o algoritmo de ABC foi capaz de obter resultados de tardeza inferiores aos de ACO, o que leva à conclusão de que para a resolução do problema de SMTWTP em situações semelhantes às do teste, e em âmbito também similar, o algoritmo de ACO é o mais adequado.

6 Conclusão

Os problemas de explosão combinatória apresentam enormes desafios na computação atual. O trabalho na eficácia da sua resolução é extremamente importante e serve de base ao crescimento e expansão de inúmeras áreas como as telecomunicações, a exploração espacial, a gestão de projeto, entre outros.

Ao realizar esta dissertação, propusemo-nos a apresentar e discutir as várias técnicas de otimização com base em *swarm intelligence* existentes na atualidade e a sua aplicabilidade na resolução de problemas de SMTWTP, fazendo uma análise de performance que nos permitisse recolher métricas relevantes, e com base nestas, retirar conclusões estatisticamente válidas. Com esse objetivo em mente, realizou-se uma análise bibliográfica dos temas e técnicas relacionadas com este paradigma que nos permitiu estabelecer qual o estado atual da arte, e de que modo as técnicas existentes se relacionam com o mundo natural de onde modelam o seu comportamento.

Adicionalmente, foi criada também uma aplicação para resolver esta família de problemas com base nos algoritmos analisados, dos quais dois foram escolhidos para o efeito. Realizou-se uma análise de performance dos algoritmos meta-heurísticos que se propõem a resolver este tipo de problema, nomeadamente o algoritmo de ACO e ABC, recorrendo à recolha de métricas da sua execução e posterior análise. Foi claro com esta análise, qual o algoritmo mais indicado para a resolução do problema de SMTWTP, tirando partido do paradigma da inteligência enxame. Os agentes artificiais simulados são capazes de comunicar entre si, direta ou indiretamente, dependendo do algoritmo, e encontrar soluções para o problema de acordo com a heurística definida.

Os resultados obtidos permitiram concluir, através de evidências estatísticas, que o número de iterações afeta de forma significativa a qualidade das soluções obtidas em ambos os algoritmos, bem como o seu tempo de execução. Foi também possível concluir que o algoritmo de ACO obtém resultados significativamente superiores na qualidade das soluções, a

custo de piores tempos de execução face ao algoritmo de ABC, indicando assim a sua superior capacidade de explorar com maior eficiência as soluções encontradas devido ao efeito da feromona sobre os outros agentes do sistema. O algoritmo de ABC é notoriamente mais indicado para situações onde a capacidade de encontrar soluções rapidamente, apesar da qualidade inferior, é o fator decisivo para uma resolução com sucesso do problema.

Todo o trabalho efetuado pretendeu contribuir, embora certamente em pequena escala, para a resolução de problemas reais de escalonamento de tarefas com as conclusões obtidas a partir dos testes efetuados.

6.1 Trabalho Futuro

A continuidade académica desta investigação inclui a realização de experiências que permitam concluir os valores ótimos para as parametrizações dos algoritmos, visto estas terem influência nos resultados da sua execução. Inclui também a realização da análise experimental com quantidades mais elevadas de tarefas a escalonar, com o objetivo de determinar qual dos algoritmos tem melhor capacidade de adaptação a ambientes de stress, em que o número de tarefas seja elevado, e em que o tempo de execução permitido ao algoritmo seja limitado, uma vez que nesta experiência apenas foi limitado o número de iterações e em vários casos de teste, nomeadamente com o aumento de tarefas a escalonar, foi possível detetar situações em que simplesmente aumentar o número de iterações não contribui necessariamente para o alcance de melhores soluções, e o custo, em forma de tempo de execução, pode ser demasiado elevado para determinadas situações.

O trabalho de continuidade será otimizar as condições de operação dos algoritmos, com o objetivo de verificar se a superioridade do algoritmo de ACO se mantém, ou se, em ambientes de teste diferentes, o algoritmo de ABC demonstra melhores resultados devido à sua capacidade de obter soluções em maior número, mais rapidamente.

Posteriormente, poderemos também realizar experiências similares com outros algoritmos de *swarm intelligence*, visto que, como foi possível concluir com base nos teste realizados, nem sempre todos os algoritmos se adequam à resolução eficiente de todas as classes de problemas, e dentro do problema de SMTWTP poderá haver algoritmos mais eficientes do que o ABC ou o ACO.

7 Referências

al-Rifaie, M., 2012. Identifying metastasis in bone scans with Stochastic Diffusion Searcg. *Proc. IEEE Information Technology in Medicine and Education*, pp. 519-523.

Beattie, P. & Bishop, J., 1998. Self-localisation in the senario autonomous wheelchair. *Journal of Intellingent and Robotic Systems,* Volume 22, pp. 255-267.

Bentley, J. L., 1992. Fast Algorythms for geometric traveling salesman problems. Em: ORSA, pp. 387-411.

Bishop, J., 1989. *Anarchic techniques for pattern classification (Ph.D. thesis),* Reading: University of Reading.

Bishop, J., 2002. Artificial Neural Networks Lecture Notes in Computer Science. *Springer,* pp. 308-313.

Bishop, J. & Torr, P., 1992. The stochastic search network. *Neural Networks for Images,* pp. 370-387.

Dorigo, M., 1992. *Optimization, Learning and natural Algorithms (Ph.D. thesis),* Italy: Politecnico di Milano.

Dorigo, M., Bonabeau, E. & Theraulaz, G., 1999. Ant Foraging Behavior. Em: *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press.

Elsevier, 2009. *Discrete Optimization.*

Faieta, B. & Lumer, E. D., 1994. Diversity and adaptation in populations of clustering ants. *Conference on Simulation of Adaptive Behavior: From Animals to Animals 3,* Issue 94, pp. 501-508.

Glover, F., 1986. Future paths for integer programming and links to artificial intelligence. pp. 533-549.

Grassé, P., 1959. La théorie de la Stigmergie: Essai dínterprétation du Comportement des Termites. *Insect. Soc.,* Volume 6, pp. 41-80.

Grech-Cini, E., 1995. *Locating facial features (Ph.D. thesis),* Reading: University of Reading.

Karaboga, D., 2012. *Artificial Bee Colony programming for symbolic regression,* Information Sciences.

Karaboga, D. & Akay, B., 2009. A comparative study of Artificial Bee Colony algorithm. *Applied Mathematics and Computation,* Issue 214, pp. 108-132.

Karaboga, D. & Akay, B., 2011. A modified Artificial Bee Colony algorithm for constrained optimization problems. *Applied Soft Computing,* Volume 11, pp. 3021-3031.

Kennedy, J. & Eberhart, R. C., 1995. Particle Swarm optimization. *Proceedings of the IEEE International Conference Neural Networks,* pp. 1942-1948.

Kube, C. R. & Bonabeau, E., 2000. Cooperative transport by ants and robots. *Autonomous Systems,* pp. 30:85-101.

Lambrinos, D., 2000. A mobile robot employing insect strategies for navigation, Robot.. *Autonomous Systems,* pp. 30:39-64.

Lewis, M., 1992. *The Bahavioral Self-Organization of Nanorobots Using Local Rules*, IEEE/RSJ Internation Conference on Intelligent Robots and Systems.

Meyer, B. & Handl, J., 2002. Improved ant-based clustering and sorting in a document retrieval interface. *Lecture Notes in Computer Science,* Volume 2439, pp. 913-923.

Pal, S., 2002. Web Mining in soft computing framework: Relevance, state of art and future directions. *IEEE Transactional on Neural Networks*, pp. 1163-1177.

Pham, D. T., 2005. *The Bees Algorithm.,* United Kingdom: Cardiff University.

Pinedo, M., 2002. Scheduling: Theory, Algorythms, and Systems, p. 3.

Potts, C. N. & Wassenhove, L. N., 1985. Em: *A branch and bound algorithm for the total weighted tardiness problem*, pp. 363-377.

Sanholzer, B. a. W., 1991. *Discrete Applied Mathematics*.

Sharma, T. K. & Bansal, J. C., 2012. *Some Modifications to Enhance the Performance of Artificial Bee Colony,* Brisbane: IEEE World Congress on Computational Inteligence.

Storn, R., 1996. *On the usage of differential evolution for function optimization.* Berkeley.

Storn, R. & Price, K., 1997. Differential Evolution - a simple and efficient heuristic for global optimization over continuous spaces.. *Journal of Global Optimization,* pp. 11:341-359.

Stutzle, T. & Hoos, H., 1997. *The MAX-MIN Ant System and Local Search for the Traveling Salesman Problem.* Los Alamitos, IEEE Computer Society Press.

University of Brunel, 2016. *OR Library.* [Online] Available at: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/ [Acedido em 1 10 2016].

Whitaker, R., 2002. *An agent based approach to site selection for wireless networks.* Madrid, IEEE Conf. on Artificial Neural Networks.

Whitaker, R., 2002. An agent based approach to site selection for wireless networks. *Proc. ACM Symposium on Applied Computing*, pp. 574-577.

8 Anexos

8.1 Manual do utilizador

8.1.1 Instalação do software necessário

Todo o software utilizado no desenvolvimento da aplicação utilizada nos testes está disponível gratuitamente.

Tendo o utilizador acesso ao código fonte da aplicação, é necessário instalar o *Microsoft Visual Studio 2015 Community Edition*, disponível no site da aplicação[2].

A instalação deve ser feita com as definições padrão, pois esta inclui todos os componentes necessários para o ambiente de desenvolvimento.

Após instalação do *Visual Studio* deve ser também instalado o *SQL Server Express*[3], igualmente com todas as definições padrão ativas. Posteriormente devem ser executados os *scripts* disponíveis no anexo 8.2, para criação da base de dados e tabelas necessárias para a correta gravação dos dados recolhidos pela aplicação.

Opcionalmente, o utilizador pode também utilizar uma instância *Azure* de um servidor *SQL Server*. Para efeitos da aplicação, a localização da base de dados é

[2] https://www.visualstudio.com/vs/community/
[3] https://www.microsoft.com/en-us/sql-server/sql-server-editions-express

irrelevante e pode ser configurada para utilizar uma instância disponível em qualquer máquina.

Após instalação o utilizador deverá dirigir-se ao local que contém o código fonte da aplicação e abrir o ficheiro SMTWTP_Solver.sln (Figura 18) e abrir o mesmo.

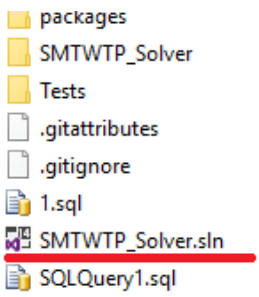

Figura 20 - Ficheiro da solução VS 2015

O primeiro passo após abertura da solução, é compilar a aplicação, para tal o utilizador deve dirigir-se ao menu *Build* e ativar a opção *Build Solution* (Figura 19).

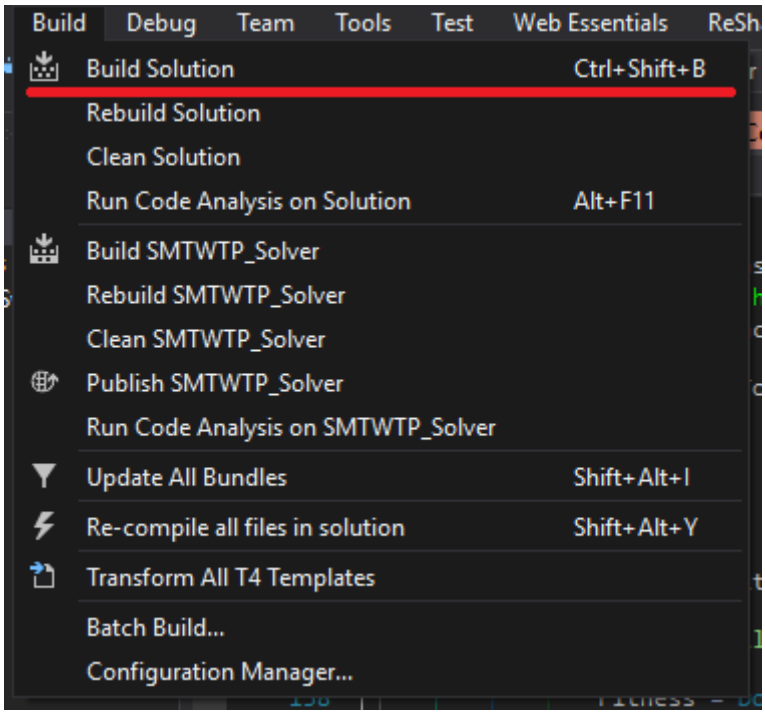

Figura 21 - Compilação da solução

Este processo irá compilar todos os ficheiros necessários e descarregar as dependências do projeto. Todo o processo não deverá demorar mais do que trinta segundos.

O passo seguinte é configurar a localização da base de dados a utilizar, para tal deverá ser expandido o projeto *SMTWTP_Solver* da solução e aberto o ficheiro *App.Config* (Figura 20).

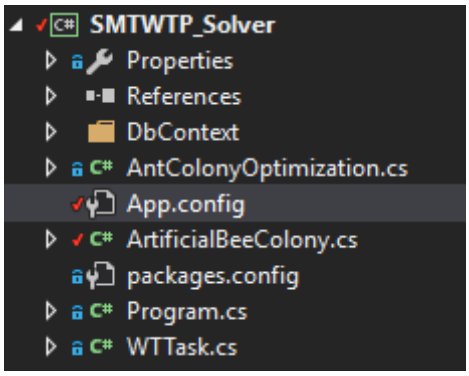

Figura 22 - App.Config

Seguidamente, dento do ficheiro, o utilizador deve localizar o elemento com o nome "*TestResults*", correspondente à *connectionString* do servidor *SQL Server* e alterar o seu valor para que utilize a *connectionString* do servidor *SQL* que o utilizador desejar.

8.1.2 Parametrização da aplicação

A parametrização a ser utilizada pelos algoritmos é também feita no ficheiro *App.Config*. As configurações gerais da aplicação incluem a variável "*Algorithm*", que define qual o algoritmo que vai executar no arranque da aplicação e deve ser configurada com os valores "ABC" para o algoritmo de *Artificial Bee Colony,* ou "ACO" para o algoritmo de *Ant Colony Optimization.* A variável "TasksFileToLoad" define qual o ficheiro da *OR Library* a carregar. "RunAlgorithmThisManyTimes" define quantas vezes o algoritmo deve executar. Finalmente, a variável "UseDB" tem como função controlar se os resultados da execução devem ou não ser gravados para a base de dados. Deve ser configurada a "*false"* caso o utilizador esteja a fazer ajustes em algum outro parâmetro e não pretenda guardar os resultados, ou se pretender apenas executar a aplicação para efeitos de teste.

```
<appSettings>
  <!--General Settings-->
  <add key="Algorithm" value="ABC" />
  <add key="TasksFileToLoad" value="wt100.txt" />
  <add key="RunAlgorithmThisManyTimes" value="100" />
  <add key="UseDB" value="false" />

  <!--Artificial Bee Colony Settings-->
  <add key="ABCMaxNoOfIterations" value="1000" />
  <add key="ABCBeePopulationSize" value="200" />
  <add key="ABCBeeFailLimit" value="20" />

  <!--Ant Colony Optimization Settings-->
  <add key="ACONumberOfAnts" value="80" />
  <add key="ACONumberOfIterations" value="1000" />
  <add key="ACOEvaporationRate" value="10" />
  <add key="ACOqValue" value="300" />
  <add key="ACOqValueMax" value="1000" />
</appSettings>
```

Figura 23 - Parameterização da aplicação

Cada um dos algoritmos tem parametrizações especificas, que o utilizador deve alterar de acordo com o teste que pretende executar. As variáveis o algoritmo de ABC são:

- ABCMaxNoOfIterations – Número de iterações a executar
- ABCBeePopulationSize – Número de abelhas
- ABCBeeFailLimit – Máximo de falhas para cada abelha

Para o algoritmo de ABC é possível configurar as seguintes variáveis:

- ACONumberOfAnts – Número de formigas
- ACONumberOfIterations - Número de iterações a executar
- ACOEvaporationRate – Rácio de evaporação da feromona
- ACOqValue – Valor mínimo da feromona
- ACOqValueMax – Valor máximo da feromona

Após realizadas as configurações desejadas, o utilizador pode executar a aplicação.

8.1.3 Recolha dos dados

O utilizador pode recolher os resultados gravados na base de dados da forma que desejar, no entanto, fica aqui definida uma opção rápida e fácil, para que não seja necessário manipular a informação através do *SQL Server.* Para tal basta abrir uma folha de cálculo no Microsoft Excel, escolher o menu *Data -> New Query -> From SQL Server Database* (Figura 22).

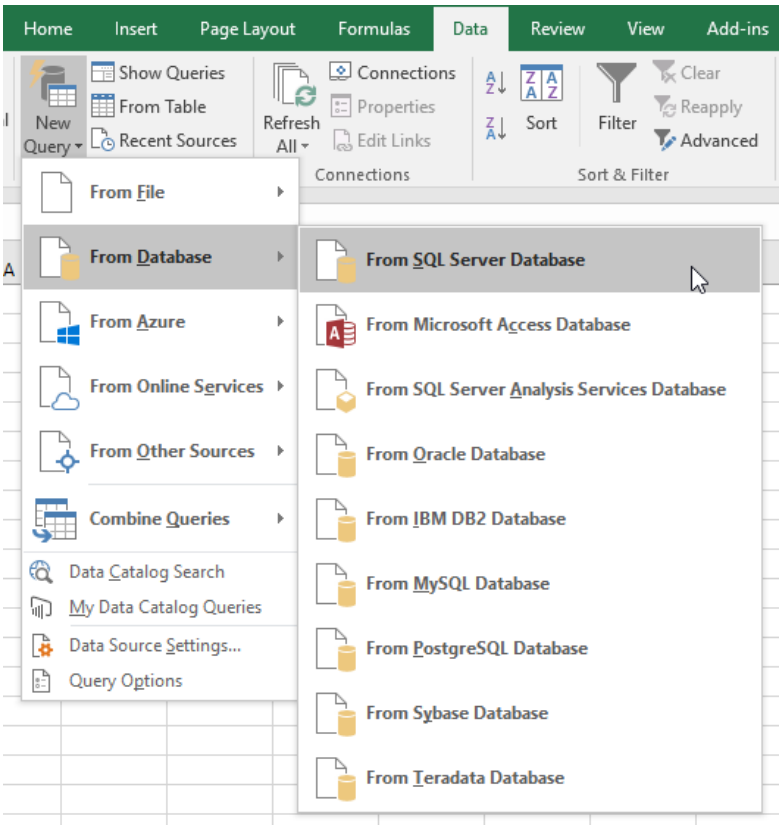

Figura 24 - Extração de dados no Excel

De seguida, deve ser inserido o IP e/ou endereço onde está alojado a servidor de *SQL Server* e depois clicar em OK (Figura 23).

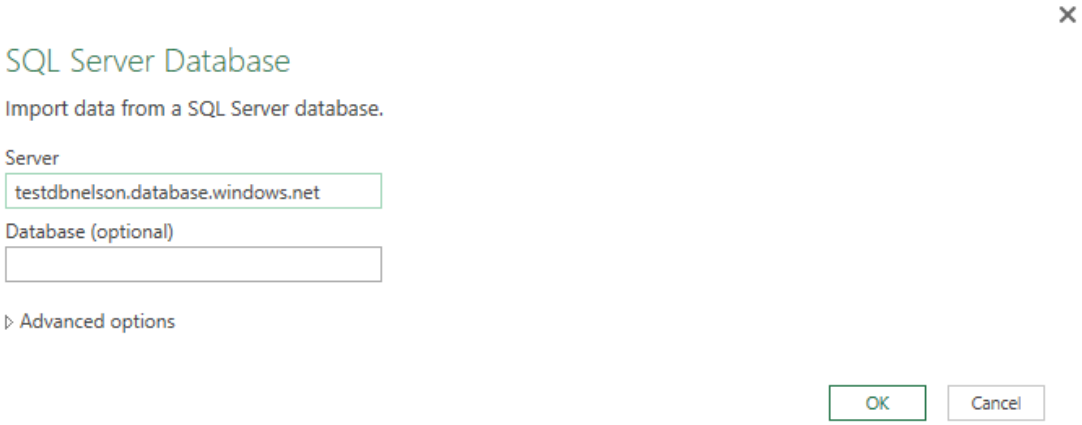

Figura 25 - Servidor *SQL Server* no excel

O Excel irá pedir as credenciais do servidor definido anteriormente, as quais o utilizador deve inserir de acordo com o que tiver configurado. Posteriormente, deve ser selecionada a tabela que contem os dados a recolher, o que aplicado ao caso desta aplicação, deve ser selecionada a tabela "*TestResults*". Posteriormente basta clicar OK e os dados serão todos carregados para a folha de Excel (Figura 24). A partir daqui os dados podem ser manipulados de acordo com a intenção do utilizador.

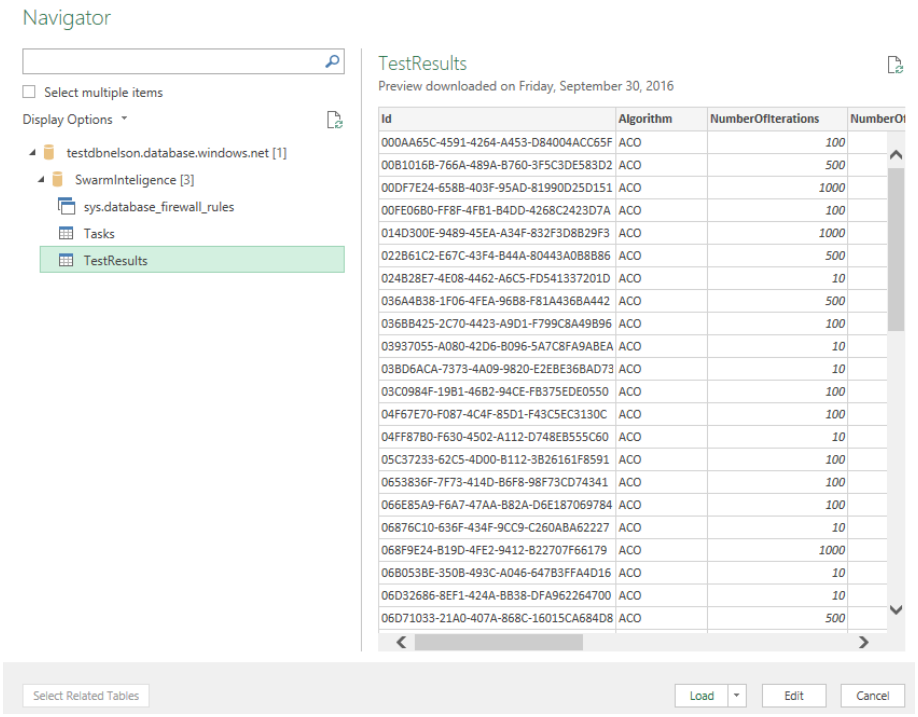

Figura 26 - Carregamento dos dados para Excel

8.2 *Script* de SQL Server

Para criação da base de dados utilizada pela aplicação deve ser executado o seguinte script na instância de *SQL Server* desejada, para criação da base e tabela que guardará os dados recolhidos pela aplicação.

```
USE [master]
GO

/****** Object:  Database [SwarmInteligence] ******/
CREATE DATABASE [SwarmInteligence]
GO

ALTER DATABASE [SwarmInteligence] SET COMPATIBILITY_LEVEL = 130
GO

USE [SwarmInteligence]

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[TestResults](
	[Id] [uniqueidentifier] NOT NULL,
	[Algorithm] [nvarchar](50) NOT NULL,
	[NumberOfIterations] [int] NOT NULL,
	[NumberOfBees] [int] NULL,
	[NumberOfAnts] [int] NULL,
	[NumberOfTasks] [int] NOT NULL,
	[ElapsedTimeInMiliseconds] [bigint] NOT NULL,
	[BestSolutionCost] [int] NOT NULL
 CONSTRAINT [PK__tmp_ms_x__3214EC07BC789410] PRIMARY KEY CLUSTERED
(
	[Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
)

GO
```

8.3 Tarefas utilizadas nos escalonamentos

8.3.1 Escalonamentos de 40 Tarefas

ExecutionTime	Weight	Deadline
69	4	0
71	1	0
93	5	0
19	5	122
71	8	660
31	6	0
28	4	818
99	6	966
21	6	249
1	8	420
89	7	152
57	8	197
87	5	0
26	7	506
94	5	593
69	10	824
1	6	719
41	3	0
86	2	0
25	9	0
69	6	0
78	8	0
49	9	241
74	5	0
43	7	0
74	8	0
47	4	0
20	3	772
41	4	801
82	9	0
37	3	0
12	8	292
97	9	814
17	7	0
63	10	469
23	9	0
10	2	430
52	10	0
44	6	42
10	8	0

8.3.2 Escalonamentos de 50 tarefas

ExecutionTime	Weight	Deadline
30	10	474
22	3	879
100	2	979
34	10	0
70	7	36
5	3	926
21	1	314
26	10	980
23	9	0
36	2	380
52	10	0
30	6	0
16	6	439
63	3	528
65	9	0
25	4	0
96	2	0
5	10	1093
57	10	115
41	8	0
94	7	268
93	2	942
45	5	0
55	5	128
92	8	0
61	2	1008
28	10	0
47	7	1059
15	1	0
38	8	826
10	9	878
67	6	166
70	4	140
28	2	0
16	2	0
5	5	0
83	2	0
79	3	0
85	8	0
15	5	309
57	2	1073
13	6	774
39	1	187
11	7	0

30	8	0
68	5	734
37	4	0
72	3	0
43	8	0
29	6	229

8.3.3 Escalonamentos de 100 Tarefas

ExecutionTime	Weight	Deadline
69	2	0
65	3	1862
33	9	926
59	8	0
40	6	513
83	2	296
92	4	0
69	1	1875
73	1	0
7	3	0
18	9	375
56	10	682
79	8	677
26	5	0
11	4	1351
3	7	0
82	1	0
24	10	1667
10	4	0
98	2	0
35	7	2051
96	5	0
74	7	0
3	8	0
87	8	0
27	10	2169
49	8	0
10	10	0
62	3	0
74	10	2547
72	5	0
93	6	0
93	7	1009
45	9	0
96	6	0
31	8	0

53	2	1047
88	8	0
11	3	247
95	3	1311
47	7	1581
71	7	0
86	5	616
22	3	0
58	9	0
37	6	1587
26	3	601
34	5	1960
100	10	0
68	3	0
64	9	1302
34	3	1894
11	2	1114
32	5	1540
87	10	364
28	1	952
17	3	0
51	7	0
34	5	637
82	9	1025
100	9	0
77	7	0
88	10	1246
24	3	1394
92	4	1872
39	8	2331
47	5	0
74	6	1409
39	3	1212
11	8	828
53	2	0
30	8	1566
71	1	1451
2	10	733
37	9	67
85	4	0
86	10	256
28	7	1694
33	10	1494
91	6	0
59	3	0
69	5	486
59	8	0

30	7	1599
44	9	1750
63	5	0
33	2	0
87	7	2546
56	2	0
11	4	981
37	2	0
85	3	0
7	2	0
92	5	0
45	10	0
68	3	1442
36	7	0
87	1	2500
23	7	2032
19	9	931

8.4 Código fonte da aplicação

O código fonte da aplicação está disponível no seguinte endereço web:

https://dl.dropboxusercontent.com/u/12294/Mestrado/SMTWTP_Solver.7z