

Resolução do problema de localização de regeneradores através de algoritmos de inteligência artificial

Pedro Ferreira

Rodrigo Pessoa

Junho 2018

Trabalho realizado no âmbito da unidade curricular de Projeto Informático

Trabalho realizado sob orientação das

Professoras Anabela Moreira Bernardino e Eugénia Moreira Bernardino.

Agradecimentos

Gostaríamos de agradecer às Professoras Anabela Moreira Bernardino e Eugénia Moreira Bernardino pelo acompanhamento no desenvolvimento do projeto. Foram verdadeiramente prestáveis e permitiram que a conclusão do projeto fosse um sucesso.

Gostaríamos também de agradecer a todos os Professores do Instituto Politécnico de Leiria pela formação de qualidade que nos prestaram ao longo do curso.

Resumo

No contexto do Projeto Informático de 2017/18 desenvolvemos uma aplicação em C++ que permite resolver o problema de localização de regeneradores.

Nas redes óticas, a força de um sinal ótico deteriora-se à medida que fica mais

longe da fonte devido a deficiências de transmissão na fibra (atenuação, dispersão, conversação). Ou seja, a distância que um sinal ótico pode ser enviado sem perder ou deteriorar a informação é limitada. É por isso necessário regenerar os sinais periodicamente utilizando regeneradores. Dada uma rede ótica, o problema de localização de regeneradores (PLR – Problema da Localização dos Regeneradores) procura que o subconjunto de regeneradores seja instalado a um custo mínimo, de modo a que cada par de nós possa comunicar entre si.

A aplicação desenvolvida por nós é bastante dinâmica tendo integradas diversas funcionalidades. Permite que cada problema seja resolvido individualmente, ilustrando graficamente a evolução do fitness do melhor individuo ao longo das gerações declaradas. Oferece também a possibilidade de resolver um conjunto especificado de problemas, utilizando sempre os mesmos parâmetros inseridos pelo utilizador, aplicando ainda 50 seeds diferentes. Por fim, esta aplicação permite também testar diferentes combinações de parâmetros de um só algoritmo, para um conjunto especificado de problemas. Os dados resultantes dos nossos testes são guardados automaticamente em formato texto e csv.

Acrescentamos que os resultados foram analisados com auxílio do programa “RapidMiner”, de forma a conseguir obter as médias de fitness, regeneradores, nós desconectados e tempo decorrido, para os diferentes seeds e instâncias dos problemas

Abstract

For our final project (2017/2018) we developed an application, built in C++, capable of solving the Regenerator Location Problem.

In optical networks, a signal’s strength deteriorates as it gets further away from its source. This happens due to deficiencies in the fiber (attenuation, dispersion, conversion). Because of this, we can say that the distance a signal can travel without losing or corrupting information is limited. It is necessary to regenerate the signals periodically using regenerators. Given an optical network, the regenerator location problem looks to install a subset of regenerators with the minimum cost possible, in a way that each pair of nodes can communicate with each other.

The developed program is quite dynamic, having integrated various functionalities. It allows for each problem to be solved individually, illustrating graphicly the evolution of the best individual’s fitness along the various generations. It also offers the possibility to solve a specified group of problems, using the same parameters declared by the user for each one, and applying 50 different seeds. The program is also cable of testing different combinations of parameters for a chosen algorithm. The resulting data from our tests is automatically stored as either a text file or a csv file.

The software “RapidMiner” was used to help in the analysis of our data by obtaining the fitness averages, regenerators, disconnected nodes, time. This for the different seeds and problem instances.

Índice

[Agradecimentos v](#_Toc517349288)

[Resumo vii](#_Toc517349289)

[Abstract ix](#_Toc517349290)

[Índice xi](#_Toc517349291)

[Lista de Figuras xiii](#_Toc517349292)

[Lista de Tabelas xv](#_Toc517349293)

[Lista de Acrónimos xvii](#_Toc517349294)

[Capítulo 1 - Introdução 1](#_Toc517349295)

[1.1 Breve Enquadramento e Descrição do Problema 1](#_Toc517349296)

[1.2 Motivação 2](#_Toc517349297)

[1.3 Estrutura do Relatório 2](#_Toc517349298)

[1.4 Síntese 3](#_Toc517349299)

[Capítulo 2 - Enquadramento 5](#_Toc517349300)

[2.1 Descrição do Problema 5](#_Toc517349301)

[2.1.1 Estrutura do problema 5](#_Toc517349302)

[2.2 Problemas de Otimização NP Hard 6](#_Toc517349303)

[2.3 Algoritmos evolutivos de inteligência artificial (algoritmos baseados em populações) 6](#_Toc517349304)

[2.4 Síntese 7](#_Toc517349305)

[Capítulo 3 – Linguagens e Tecnologias 9](#_Toc517349306)

[3.1 C++/CLI 9](#_Toc517349307)

[3.1.1 A implementação com C++/CLI 9](#_Toc517349308)

[3.2 Qt 10](#_Toc517349309)

[3.2.1 A implementação com a framework Qt 10](#_Toc517349310)

[3.3 Git e Github 12](#_Toc517349311)

[3.4 RapidMiner 12](#_Toc517349312)

[3.5 Síntese 12](#_Toc517349313)

[Capítulo 4 - Aplicação 15](#_Toc517349314)

[4.1 Análise de Requisitos 15](#_Toc517349315)

[4.2 Metodologia 15](#_Toc517349316)

[4.2.1 As Diferentes Etapas de Desenvolvimento 16](#_Toc517349317)

[4.2.2 Metodologia de Testes 16](#_Toc517349318)

[4.3 Estrutura do Programa 17](#_Toc517349319)

[4.3.1 Classes 17](#_Toc517349320)

[4.3.1.1 Algorithm 18](#_Toc517349321)

[4.3.1.2 Dialog 19](#_Toc517349322)

[4.3.1.3 MultiThread 19](#_Toc517349323)

[4.3.1.4 TestDialog 20](#_Toc517349324)

[4.3.1.5 TestMultiThread 20](#_Toc517349325)

[4.3.1.6 Thread 21](#_Toc517349326)

[4.3.1.7 Individual 21](#_Toc517349327)

[4.3.1.8 MainWindow 22](#_Toc517349328)

[4.3.1.9 Population 22](#_Toc517349329)

[4.3.1.10 Problem 23](#_Toc517349330)

[4.4 Algoritmos 24](#_Toc517349331)

[4.4.1 Custom Algorithm 24](#_Toc517349332)

[4.4.2 Algoritmo Genético 24](#_Toc517349333)

[4.4.2.1 Elitismo 25](#_Toc517349334)

[4.4.2.2 Torneio 25](#_Toc517349335)

[4.4.2.3 Recombinação com um corte 26](#_Toc517349336)

[4.4.2.4 Mutação 26](#_Toc517349337)

[4.4.3 Bee Colony Optimization 26](#_Toc517349338)

[4.4.4 Ant Colony Optimization 26](#_Toc517349339)

[4.4.4.1 Inicialização de parâmetros 27](#_Toc517349340)

[4.4.4.2 Função de Avaliação 27](#_Toc517349341)

[4.4.4.3 Inicialização do trilho de feromonas 27](#_Toc517349342)

[4.4.4.4 Clonar a atual formiga para um individuo auxiliar 28](#_Toc517349343)

[4.4.4.5 Testa probabilidade exploração 28](#_Toc517349344)

[4.4.4.6 Testa intensificação 28](#_Toc517349345)

[4.4.4.7 Busca Local 28](#_Toc517349346)

[4.4.4.8 Atualiza o trilho de feromonas 28](#_Toc517349347)

[4.4.4.9 Notas 28](#_Toc517349348)

[4.5 Síntese 29](#_Toc517349349)

[Capítulo 5 - Resultados 31](#_Toc517349350)

[5.1 Obtenção de dados 31](#_Toc517349351)

[5.1.1 Teste de parâmetros dos algoritmos 31](#_Toc517349352)

[5.1.2 Recolha de dados de execução dos algoritmos 31](#_Toc517349353)

[5.2 Tratamento dos dados obtidos 31](#_Toc517349354)

[5.2.1 Tratamento dos dados de teste de parâmetros 31](#_Toc517349355)

[5.2.2 Tratamento dos dados de execução dos algoritmos 31](#_Toc517349356)

[5.3 Síntese 32](#_Toc517349357)

[Capítulo 6 - Conclusão 33](#_Toc517349358)

[6.1 Comparação dos algoritmos implementados 33](#_Toc517349359)

[6.2 Escolha dos melhores algoritmos 33](#_Toc517349360)

[6.3 Trabalho futuro 33](#_Toc517349361)

[6.4 Síntese 33](#_Toc517349362)

[Bibliografia 35](#_Toc517349363)

Lista de Figuras

[Figura 1 - Programa em C++/CLI 8](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861410)

[Figura 2 - Programa em Qt 10](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861411)

[Figura 3 - Desenvolvimento Iterativo e Incremental 12](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861412)

[Figura 4 - Custom Dialog 15](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861413)

[Figura 5 - Ant Colony Test Dialog 16](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861414)

[Figura 6 - Main Window 19](file:///E:\Git\RLP_Qt\Relatório-2151163-2150662.docx#_Toc516861415)

Lista de Tabelas

<criar automaticamente a lista de tabelas>

Lista de Acrónimos

PLR Problema de Localização de Regeneradores

AG Algoritmo Genético

ACO Ant Colony Optimization

BCO Bee Colony Optimization

CA Custom Algorithm

1. - Introdução

Este capítulo está estruturado da seguinte forma: A secção 1.1 apresenta o enquadramento e a descrição do problema resolvido neste projeto; A secção 1.2 apresenta a motivação para o desenvolvimento deste projeto; A secção 1.3 apresenta a estrutura deste relatório, e a secção 1.4 sumariza os conceitos apresentados neste capítulo.

* 1. Breve Enquadramento e Descrição do Problema

Na unidade curricular de Projeto Informático temos de nos candidatar a algumas das propostas disponibilizadas, para que depois possam ir a concurso e ser distribuídas entre os alunos. Discutimos entre nós quais seriam as propostas a que nos iriamos candidatar, tendo acordado em colocar o PLR como principal opção.

A Inteligência Artificial tem vindo a ganhar imensa popularidade ultimamente, devido às suas capacidades de permitir que máquinas aprendam e se ajustem a diferentes problemas. Tem já uma grande presença no nosso dia a dia, sendo aplicada muito frequentemente em jogos de computador, automóveis, assistentes digitais (como a *Amazon Echo*), robots, etc. No núcleo de toda a inteligência artificial estão os seus algoritmos, pois são estes que treinam a inteligência para que esta aprenda. No caso do PLR, o requisito mínimo era a implementação de dois algoritmos de inteligência artificial para resolver o problema, tendo o nosso grupo implementado quatro.

O PLR é um problema real, pois de facto existe um problema nas redes óticas. Quanto mais distante estiver o sinal ótico da sua fonte, mais sujeito este está a perder qualidade. Isto acontece devido a falhas na fibra em si. De modo a resolver este problema, existem máquinas chamadas regeneradores, cujo propósito é restaurar a qualidade do sinal. Infelizmente, estes equipamentos tem um custo bastante elevado.

O nosso objetivo é resolver o PLR, obtendo soluções com o menor custo possível, no menor tempo possível. Todo este processo foi feito recorrendo a diferentes algoritmos de Inteligência Artificial. Os algoritmos que escolhemos utilizar foram

o CA, o AG, o ACO e o BCO.

Por fim, comparamos o desempenho dos algoritmos, tendo determinado o melhor.

* 1. Motivação

Com a evolução das tecnologias de comunicação, existe cada vez mais a necessidade por um serviço rápido, seguro e sem falhas. Como resposta a esta necessidade surgem as redes de fibra ótica.

Como mencionado previamente, o PLR é um problema do mundo real. Decidimos encará-lo pela sua relevância e pelo desafio de encontrar uma solução tão boa ou melhor que as já existentes.

* 1. Estrutura do Relatório

Este relatório está estruturado da seguinte maneira:

* Capítulo 1 - Introdução

Neste capítulo é feita a apresentação do nosso problema e trabalho, para que o leitor seja contextualizado e ganhe uma ideia geral dos temas que serão tratados ao longo do relatório. Resumimos todo o projeto e explicamos as nossas motivações para este trabalho.

* Capítulo 2 – Enquadramento

Neste capítulo entramos em detalhe sobre o nosso problema, explicando-o detalhadamente e fornecendo vários conceitos importantes para a compreensão do mesmo.

* Capítulo 3 – Linguagens e tecnologias

Neste capítulo falamos sobre a linguagem de programação aplicada e das tecnologias que tiveram mais impacto neste projeto, como o QT e o GitHub. Falamos também das várias dificuldades que tivemos, nomeadamente a adaptação a uma linguagem de programação com que nunca tínhamos lidado.

* Capítulo 4 – Aplicação

Este é o nosso capítulo mais extenso, pois é onde entramos em detalhe relativamente à aplicação. Começamos por detalhar quais os requisitos impostos para este projeto, falando de seguida na metodologia de trabalho aplicada por nós. Falamos também da estrutura do nosso programa e por fim dos algoritmos aplicados.

* Capítulo 5 – Resultados

Neste capítulo falamos sobre os resultados obtidos, mais concretamente, como foram obtidos e como foram tratados.

* Capítulo 6 - Conclusão

Neste capítulo fazemos o estudo comparativo determinando a nossa melhor solução para o problema.

* 1. Síntese

Com este capítulo foi possível identificar os pontos mais importantes deste relatório, bem como de todo o projeto em si. Foi brevemente explicado o problema e como este foi resolvido. Deixamos clara a nossa motivação para trabalhar neste projeto e definimos a estrutura de todo o relatório.

1. - Enquadramento

Este capítulo está estruturado da seguinte forma: A secção 2.1 apresenta a descrição do problema resolvido neste projeto; A secção 2.2 apresenta o conceito de problemas otimização do tipo NP Hard; A secção 2.3 apresenta os tipos de algoritmos que permitem a solução deste tipo de problema, e a secção 2.4 sumariza os conceitos apresentados neste capítulo.

* 1. Descrição do Problema

O problema pretende garantir que numa rede de fibra ótica os sinais são regenerados de forma a manter a qualidade do sinal, que se deteriora proporcionalmente à distância que percorre na rede.

Os problemas utilizados no nosso projeto estão presentes no website do centro de computadores da universidade de Viena, que também abordou previamente o PLR. Estes problemas são representações simplificadas de redes de fibra ótica, contendo um conjunto de nós interligados. Para encontrar a solução deste problema é necessário conseguir descobrir uma disposição de regeneradores de sinais num conjunto de nós interligados, de forma a que todos os nós contenham ou estejam ligados a pelo menos um regenerador. Este processo terá de ser feito utilizando o menor número de regeneradores possível. No caso de problemas em que a colocação de regeneradores num nó específico tem custos diferentes, o objetivo é ter o menor custo possível.

A aplicação ao mundo real da nossa forma de resolver o problema necessitaria de uma ferramenta capaz de converter uma rede de fibra ótica real numa rede simplificada como as que foram utilizadas.

* + 1. Estrutura do problema

Os problemas utilizados estão inicialmente sob o formato de ficheiros de texto, sendo estes obtidos do site do “Centro de Computadores da Universidade de Viena”. A principal informação contida nestes ficheiros é uma matriz que representa as ligações entre os diferentes nós que compõem o problema. Para alem disto, contêm também o número de nós e o número de ligações.

Alguns dos problemas contêm também um vetor com pesos, referente ao custo da colocação de um regenerador em cada nó específico.

* 1. Problemas de Otimização NP Hard

Um problema é classificado como NP-Hard (***N****on-deterministic* ***P****olynomial time*) quando pode ser considerado pelo menos tão difícil quanto o problema mais difícil do tipo NP-Hard. Por outras palavras, qualquer algoritmo utilizado para resolver um problema do tipo NP-Hard em tempo polinomial pode ser posteriormente traduzido para resolver qualquer problema também NP-Hard.

Problemas do tipo NP-Hard podem ser: problemas de decisão, problemas de pesquisa ou problemas de otimização.

* 1. Algoritmos evolutivos de inteligência artificial (algoritmos baseados em populações)

Os Algoritmos Evolutivos têm como objetivo trabalhar com o melhor de uma solução, e melhorá-la. Esta ideia é baseada nos princípios da teoria da evolução biológica, isto é, pode-se simular o processo evolutivo da natureza de modo a encontrar soluções ótimas para os nossos problemas. Deste princípio, surgiram vários algoritmos como por exemplo o Algoritmo Genético (AG).

Por norma, todos os algoritmos evolutivos começam com uma população inicial. Desta população, cada elemento (ou individuo) representa uma solução para o problema. Podemos deduzir que nem todos indivíduos são ótimos, sendo necessária uma avaliação. Desta avaliação irá resultar o fitness**,** que representa a qualidade de um individuo como solução para o nosso problema. O próximo passo será comparar os fitnesses, identificando os melhores indivíduos para que sejam preservados e passem para a próxima geração, da qual irão ser membros e serão utilizados como referência para gerar novos indivíduos. Através deste processo onde se tira proveito das qualidades dos melhores indivíduos para gerar soluções melhores ao longo das várias gerações, irá resultar uma solução final que em princípio será a ótima para o problema. Todo este processo de procura pela melhor solução repete-se várias vezes durante um certo número de iterações (chamadas de gerações) até que uma condição seja alcançada, como por exemplo se “um objetivo foi cumprido” ou se “foi atingido o número máximo de gerações”.

* 1. Síntese

Com este capítulo explicamos detalhadamente o principal tema deste projeto, o problema da localização de regeneradores, bem como alguns conceitos pertinentes para a compreensão deste, como é o caso de NP-Hard e o que são algoritmos evolutivos.

1. – Linguagens e Tecnologias

Este capítulo está estruturado da seguinte forma: A secção 3.1 apresenta a linguagem de programação utilizada no desenvolvimento deste projeto; A secção 3.2 apresenta a framework utilizada no desenvolvimento deste projeto; A secção 3.3 apresenta a ferramenta Git e a plataforma GitHub, utilizadas para o controlo de versões no desenvolvimento deste projeto; A secção 3.4 apresenta a ferramenta de *data mining* RapidMiner, e a secção 3.5 sumariza os conceitos apresentados neste capítulo.

* 1. C++/CLI

Inicialmente foi discutido o programa ser desenvolvido em C++ devido à rapidez de execução, que se revela importante na solução deste tipo de problemas. Já estávamos familiarizados com a linguagem C contudo foi necessário adaptarmo-nos à linguagem C++.

C++/CLI ou C++ modificado para “*Common Language Infrastructure*” é a integração de C++ com “Windows Forms”. Contudo, esta implementação é quase como uma outra linguagem de programação, diferente de C++. Grande parte das instruções não podem ser executadas, havendo grandes mudanças na sintaxe e diferenças nos objetos.

Inicialmente, o programa foi implementado em C++/CLI, contudo, devido às limitações que acompanhavam esta plataforma (software Visual Studio 2017), acabamos por optar pela framework de C++ “Qt”.

* + 1. A implementação com C++/CLI

Com C++/CLI, foi implementada uma janela com os parâmetros do algoritmo e 2 botões, um botão para ler um problema outro para resolver o problema.

A leitura do problema e feita através da leitura de uma matriz de ligações entre nós, presente nos diferentes ficheiros de texto. Alguns ficheiros têm também pesos associados a cada um dos nós, onde cada peso representa o custo de colocar um regenerador nesse nó.

A resolução do problema aplica o algoritmo que denominamos de “Custom Algorithm” (cujos detalhes estão mais à frente explicados no capítulo 4). Para tal, uma nova thread é criada. Esta vai aplicando o algoritmo e a população vai evoluindo. A cada geração, a thread invoca um método do form que o atualiza, para que se mostrem os dados do melhor individuo da população atual, bem como um gráfico que retrata a evolução do fitness ao longo das várias iterações do algoritmo.

Figura 1 - Programa em C++/CLI

* 1. Qt

Qt é uma ferramenta de desenvolvimento para desktop compatível com diferentes plataformas. Não é só por si uma linguagem de programação, mas é uma framework escrita em C++ com características adicionais como *signals and slots*, e o funcionamento do seu MOC (Meta-Object Compiler) permite converter todo o código escrito de forma a ser compilado por qualquer compilador de C++.

* + 1. A implementação com a framework Qt

O programa desenvolvido em Qt é a versão final do nosso projeto.

Esta versão contém uma janela principal com a opção de selecionar um dos quatro algoritmos implementados, e ainda uma ferramenta extra para ordenar os resultados guardados nos ficheiros csv.

Cada algoritmo tem a sua janela própria, em que temos os parâmetros requeridos pelo algoritmo em questão, um botão para resolver um problema individualmente, um botão para resolver uma série de problema, e um botão para testar combinações de parâmetros.

Na solução de um problema individual, é selecionado o ficheiro do problema, este é integrado na classe problema, e de seguida é criada uma thread especifica ao algoritmo selecionado, que a cada geração envia um sinal à thread da interface visual, com a informação do melhor individuo da geração atual, de forma a atualizar os dados da janela e do gráfico da evolução do fitness ao longo das gerações.

Na solução de um conjunto de problemas, é selecionada uma pasta com diferentes problemas, é criado um ficheiro csv onde são guardados os resultados da execução do algoritmo para cada um dos ficheiros e para cada seed diferente. São criadas várias threads, e cada thread vai resolvendo um problema diferente das restantes threads, para tal é lido o problema, criada a população, é aplicado o algoritmo, e no fim de obter a solução, é enviado um sinal à thread original para escrever os resultados no ficheiro csv.

No teste de diferentes combinações de parâmetros, é selecionada uma pasta com os problemas a testar, é introduzido uma combinação de valores a testar para cada parâmetro, é criado um ficheiro csv para os resultados da execução. São criadas várias threads, e cada thread resolve um problema diferente, e aplica ao mesmo problema as diferentes combinações de parâmetros, sempre que a solução para um conjunto de parâmetros termina, é enviado um sinal à thread original para escrever os resultados no ficheiro csv.

Figura 2 - Programa em Qt

* 1. Git e Github

Com o auxílio da ferramenta de controlo de versões Git, foi utilizada a plataforma GitHub para hospedar os repositórios utilizados durante o desenvolvimento do projeto.

À medida que eram feitos progressos no desenvolvimento da aplicação estes eram submetidos para a plataforma, apenas foi utilizado o branch Master pois nunca chegou a haver conflitos sendo que o trabalho desenvolvido foi sempre repartido de forma a evitar conflitos.

* 1. RapidMiner

Com o auxílio deste software de *data science*, foi possível partir de uma grande quantidade de dados em bruto e obter informação relevante sobre as diferentes combinações de parâmetros para os algoritmos bem como condensar a informação referente à execução dos algoritmos para um grande número de problemas e seeds.

* 1. Síntese

Neste capítulo explicamos quais as linguagens/software com que trabalhamos durante o desenvolvimento deste projeto, tendo referido a necessidade de adaptação a uma nova linguagem e framework. Apesar da não familiaridade com estes tópicos, fomos capazes de provar a nossa capacidade para adaptação, tendo no longo-prazo conseguido alcançar o sucesso deste projeto.

1. - Aplicação

Este capítulo está estruturado da seguinte forma: A secção 4.1 apresenta a análise de requisitos efetuada para o desenvolvimento deste projeto; A secção 4.2 apresenta a metodologia utilizada no desenvolvimento deste projeto; A secção 4.3 apresenta a estrutura do programa desenvolvido, e a secção 4.4 sumariza os conteúdos apresentados neste capítulo.

* 1. Análise de Requisitos

Os requisitos necessários para o desenvolvimento do programa foram os seguintes:

* Leitura de problemas a partir de ficheiros de texto;
* Criação de um conjunto de soluções para o problema;
* Melhorar as soluções propostas com pelo menos dois algoritmos evolutivos;
* Observar a evolução das soluções em tempo real;
* Otimização do funcionamento dos algoritmos;
* Resolver múltiplos problemas em simultâneo;
* Guardar dados sobre as soluções para serem analisados;
  1. Metodologia

A metodologia adotada no desenvolvimento deste projeto foi a de “desenvolvimento iterativo e incremental”. A cada etapa foram designados objetivos, em seguida foi feito o levantamento de requisitos, seguido da implementação das funcionalidades, e por fim o teste de cada funcionalidade nova implementada.

* + 1. As Diferentes Etapas de Desenvolvimento

Figura 3 - Desenvolvimento Iterativo e Incremental

A primeira etapa consistiu em implementar a leitura de problemas a partir de ficheiros de texto. Na segunda etapa foram criadas as classes Individual e Population, bem como a fórmula para o cálculo do fitness de um individuo. A terceira etapa consistiu na implementação do nosso primeiro algoritmo evolutivo (Custom Algorithm), bem como a forma de resolver os problemas utilizando o algoritmo implementado. Na quarta etapa foi implementada a forma de resolver múltiplos problemas em simultâneo em threads diferentes. Na quinta etapa foram implementados os recursos necessários para obter a melhor combinação de parâmetros para cada algoritmo. Na sexta e última etapa foram implementados os algoritmos evolutivos restantes.

* + 1. Metodologia de Testes

Foram efetuados dois tipos de testes ao software, testes para validar o funcionamento correto, e testes de performance.

De forma a validar o funcionamento correto do programa foi utilizada a ferramenta de debugging presente no IDE utilizado (*Qt Creator*).

De forma a testar a performance do programa foi utilizado o primeiro algoritmo implementado como base de comparação.

* 1. Estrutura do Programa

Com este capítulo espera-se dar um entendimento de como o software funciona, a utilização típica do programa consiste em selecionar um algoritmo, introduzir os parâmetros para o algoritmo, e em seguida resolver um ou vários problemas, sendo que os resultados do problema individual são apresentados no programa, enquanto que os resultados da solução uma série de problemas podem ser consultados num ficheiro csv.

* + 1. Classes

As classes que compõem o programa são as seguintes:

* AntColonyAlgorithm
* AntColonyDialog
* AntColonyMultiThread
* AntColonyTestDialog
* AntColonyTestMultiThread
* AntColonyThread
* BeeColonyAlgorithm
* BeeColonyDialog
* BeeColonyMultiThread
* BeeColonyTestDialog
* BeeColonyTestMultiThread
* BeeColonyThread
* CustomAlgorithm
* CustomDialog
* CustomMultiThread
* CustomTestDialog
* CustomTestMultiThread
* CustomThread
* GeneticAlgorithm
* GeneticDialog
* GeneticMultiThread
* GeneticTestDialog
* GeneticTestMultiThread
* GeneticThread
* Individual
* MainWindow
* Population
* Problem

A razão para haver tantas classes parecidas para algoritmos diferentes é que cada algoritmo tem diferentes parâmetros, pelo que optámos por ter uma janela diferente para cada algoritmo ao invés de uma janela dinâmica que se altera conforme é selecionado um algoritmo especifico, e também porque em C++ o polimorfismo não é fácil de implementar devido ao *splicing* de objetos, o polimorfismo foi algo em que investimos imenso tempo a tentar implementar mas nunca conseguimos os resultados esperados, contudo esta solução comporta-se da maneira desejada.

Nas próximas secções está a descrição das diferentes classes, algumas foram agrupadas devido à semelhança a outras classes.

* + - 1. Algorithm

As classes Algorithm (AntColonyAlgorithm, BeeColonyAlgorithm, CustomAlgorithm, GeneticAlgorithm) contêm os seguintes métodos principais:

**setUpAlgorithm** **-** Funciona como o construtor da classe, o construtor está vazio de forma a estas classes poderem ser instanciadas sem argumentos, e com esta função a mesma instância pode ser utilizada para uma diferente configuração de parâmetros.

**generateNewPopulation** **-** É a função que contem o funcionamento principal do algoritmo, conforme o funcionamento deste, será gerada uma nova população (solução), com base na população anterior. No capítulo 2 está em detalhe o funcionamento dos diferentes algoritmos pelo que será omitido nesta fase do relatório.

Para além destes métodos principais, certos algoritmos têm ainda métodos adicionais, que auxiliam o método generateNewPopulation.

* + - 1. Dialog

Cada Dialog (AntColonyDialog, BeeColonyDialog, CustomDialog, GeneticDialog) é composto por uma série de inputs e labels correspondentes aos diferentes parâmetros que o algoritmo necessita para correr, um *drop down* com o número de threads a utilizar e um *drop down* com o intervalo de seeds a executar na solução de múltiplos problemas. Os Dialogs têm também 3 botões cada um com uma função diferente, tendo estes os labels “Solve”, “Batch Solve” e “Test”. Para além de inputs e botões, os Dialogs têm também labels para informações do problema atual, uma zona chamada de GridLayout que contem ou um gráfico do fitness em relação às gerações ou um conjunto de labels com informação sobre as diferentes threads em execução, e uma barra de progresso representativa da execução atual do algoritmo.

Figura 4 - Custom Dialog

* + - 1. MultiThread

As classes MultiThread (AntColonyMultiThread, BeeColonyMultiThread, CustomMultiThread, GeneticMultiThread) contêm os seguintes métodos principais:

* run
* newProblem
* problemEnded

run é a função principal da thread, é esta que é executada quando se faz thread->start() dentro da thread original, e o que faz é para cada problema dentro da pasta selecionada, se este for um dos problemas que esta thread tem de tratar, aplica o algoritmo ao mesmo problema com 10 seeds diferentes, e envia os resultados de cada solução para a thread original.

newProblem é um sinal que é enviado à thread original quando a MultiThread começa a tratar um novo problema, é enviado o número da thread e o nome do problema que está a tratar, para ser mostrado no *grid layout* da janela.

problemEnded é um sinal que é enviado à thread original quando a MultiThread acaba de tratar uma seed para um problema, é enviada a *string* formatada de forma a ser escrita no ficheiro csv que está aberto na thread original.

* + - 1. TestDialog

As classes TestDialog (AntColonyTestDialog, BeeColonyTestDialog, CustomTestDialog, GeneticTestDialog) consistem em janelas com labels e inputs com valores de inicio, fim e incrementos para os diferentes parâmetros a variar, todos estes inputs têm *getters* a ser chamados pela janela original de forma a obter os valores introduzidos, e estes são utilizados para construir as classes TestMultiThread.

Figura 5 - Ant Colony Test Dialog

* + - 1. TestMultiThread

As classes TestMultiThread (AntColonyTestMultiThread, BeeColonyTestMultiThread, CustomTestMultiThread, GeneticTestMultiThread) contêm os mesmos métodos principais que as classes MultiThread, contudo a sua função *run* ao invés de aplicar diferentes seeds ao mesmo problema, aplica diferentes combinações de parâmetros para o algoritmo.

* + - 1. Thread

As classes Thread (AntColonyThread, BeeColonyThread, CustomThread, GeneticThread) existem com o intuito de a janela original poder ser atualizada à medida que um algoritmo é aplicado a um problema. Contêm os seguintes métodos principais:

* Run
* dataChanged
* singleProblem

run é a função principal da thread, é esta que é executada quando se faz thread->start() na thread original, a função instancia o problema, a população e o algoritmo, notifica a thread original dos dados do problema, e à medida que o algoritmo vai gerando novas populações, notifica a thread original com os dados do melhor individuo da população atual.

dataChanged é o sinal que é enviado a cada iteração do algoritmo, enviando todos os dados necessários para atualizar a interface (fitness, nós desconectados, regeneradores, geração e progresso).

singleProblem é o sinal que é enviado quando um novo problema é iniciado, este contem os dados do problema e da população inicial (fitness, nós desconectados, regeneradores, geração, total de nós e total de ligações).

* + - 1. Individual

A classe Individual representa uma solução do problema, o seu construtor recebe o problema e prioritiza colocar regeneradores nos nós com mais ligações, a classe tem os seguintes atributos:

* QVector<int> solution
* int fitness
* int disconnected
* int regenerators

solution é um vetor de inteiros, com tamanho N sendo N o número de nós no problema, cada índice do vetor toma o valor 1 ou 0 conforme esse nó tem ou não um regenerador.

fitness é o valor inteiro obtido do calculo do fitness do individuo.

disconnected é o valor inteiro do número de nós desconectados na solução.

regenerators é o valor inteiro do total de regeneradores na solução.

A classe Individual tem os seguintes métodos principais:

* calculateFitness
* clone

calculateFitness é uma função que verifica cada ligação do problema conforme a solução apresentada, e por cada nó desconectado é somado 500 ao valor do fitness, por cada regenerador utilizado é somado 100 ao valor do fitness, ou 100 vezes o peso do nó para os problemas com pesos.

clone é uma função que cria uma nova instância da classe Individual, mas com novas referencias de memoria.

* + - 1. MainWindow

A classe main window apenas contem quatro botões que iniciam uma nova janela com o algoritmo selecionado, e uma ferramenta que aplica a função sort, que o que faz é ler um ficheiro de resultados gerados pelo programa, e ordena-os de forma a ficar um ficheiro de mais fácil interpretação.

Figura 6 - Main Window

4.3.1.9 Population

O principal atributo da classe Population é QVector<Individual> individuals, que é um vetor de indivíduos.

Os principais métodos da classe Population são:

* setUpPopulation
* calculateFitnesses
* getBestIndividual

setUpPopulation permite criar uma nova população sem ter de criar uma nova instância da classe, e permite também instanciar a classe sem saber os parâmetros inicialmente.

calculateFitnesses funciona da mesma forma que a função calculateFitness da classe Individual, contudo a utilização desta função torna-se mais rápida do que executar individualmente para cada individuo e tem a vantagem de ordenar os indivíduos por ordem de fitness, pelo que acaba por ser mais utilizada que a anterior.

getBestIndividual é uma função que só deve ser chamada depois de os fitnesses terem sido calculados e ordenados, pois esta retorna o primeiro Individual da variável individuals.

* + - 1. Problem

Os atributos da classe Problem são os seguintes:

* QVector<QVector<int>> nodes
* QVector<int> weights
* QVector<float> connectionsWeight
* int hasWeight
* int total
* int connections

nodes é uma matriz de inteiros que contem as ligações entre os nós do problema, ou seja, sendo a coluna x e a linha y, se o elemento da matriz na posição x, y tiver o valor 1, significa que o nó no índice x está ligado ao nó no índice y, como tal é uma matriz simétrica.

weights é um vetor de inteiros que caso o problema tenha pesos, cada índice contém o peso de colocar um regenerador no nó desse índice.

connectionsWeight é um vetor que toma valores entre 0 e 1, sendo 0 um nó que não tem ligações, e 1 o nó com mais ligações.

hasWeight é um inteiro que toma o valor 1 quando o problema tem pesos.

total é o número de nós do problema.

connections é o número de ligações entre nós.

O principal método da classe Problem é setUpProblem, que permite criar um novo problema sem ter de criar uma nova instância da classe, e que permite o objeto ser instanciado sem saber o argumento necessário. Este método recebe o caminho para o ficheiro do problema a ler, e desse ficheiro obtém o tamanho do problema, o número de ligações, os pesos dos nós (se existirem) e a matriz de ligações entre nós.

* 1. Algoritmos

Todos os algoritmos necessitam de uma população inicial, esta não é gerada de forma totalmente aleatória, mas sim tendo em conta os nós com mais ligações, tendo maior probabilidade de colocar um regenerador nesses nós.

Todos os algoritmos são inicializados de forma diferente, mas o seu funcionamento principal passa pela execução da função generateNewPopulation até chegar à geração final introduzida nos parâmetros.

* + 1. Custom Algorithm

O CA requer uma população inicial. A função de cálculo de fitness é aplicada a cada geração, e para além de calcular o fitness dos indivíduos também os ordena em ordem crescente de fitness.

Os parâmetros mais relevantes para este algoritmo são:

* Elitismo
* Probabilidade de mutação

O algoritmo funciona do seguinte modo:

Inicialização de parâmetros

Enquanto não é atingida a condição de termino

Cria uma nova população com estes passos:

* + - Elitismo
    - Recombinação
    - Mutação
    - Recombinação 2

Calcula fitness e ordena a nova população

Fim do ciclo

De seguida estão explicados os passos do algoritmo:

4.4.1.1 Elitismo

O elitismo é um parâmetro de entrada que age como uma forma de seleção. Aplicamos o elitismo à população presente no inicio de cada iteração do algoritmo de modo a manter os seus melhores indivíduos (no nosso caso temos a população organizada de melhor para pior). Ou seja, tendo uma população de 100 indivíduos e um elitismo de 20%, irão ser mantidos os primeiros 20 indivíduos. Estes indivíduos não são alterados na iteração atual.

4.4.1.2 Recombinação

Os restantes indivíduos não selecionados pelo elitismo são completamente refeitos, através de uma combinação dos diferentes genes dos indivíduos selecionados por elitismo, sendo que o gene na posição n do individuo a recombinar, toma o valor do gene na posição n de um dos indivíduos selecionados por elitismo.

4.4.1.3 Mutação

De forma a saber onde aplicar mutações, é feito o cálculo para cada gene dos indivíduos selecionados por elitismo, para averiguar se algum gene apenas tem um valor, e caso seja verdade, a primeira metade dos indivíduos não selecionados, segundo a probabilidade de mutação, podem ter o gene nessa posição alterado para o valor que não existe nos indivíduos selecionados.

A segunda metade dos indivíduos não selecionados, segundo a probabilidade de mutação, pode ter qualquer gene alterado para um valor aleatório dentro dos valores considerados no problema (0 e 1).

4.4.1.4 Recombinação 2

A segunda metade dos indivíduos selecionados são completamente refeitos, através de uma combinação dos diferentes genes da primeira metade dos indivíduos selecionados por elitismo, sendo que o gene na posição n do individuo a recombinar, toma o valor do gene na posição n de um dos indivíduos da primeira metade dos indivíduos selecionados por elitismo.

* + 1. Algoritmo Genético

O AG requer uma população inicial. Cada individuo desta população é avaliado utilizando a função de avaliação, cujo resultado deverá representar a qualidade do individuo como solução para o problema. Através do método de seleção elitismo e torneio, iremos obter uma nova população. De seguida, são aplicados os operadores genéticos de recombinação e mutação, respetivamente, com intuito de gerar novos indivíduos. Com cada geração avaliamos a nova população gerada. No final de todo o processo evolutivo o algoritmo devolve o melhor individuo encontrado.

Os parâmetros mais relevantes para este algoritmo são:

* Probabilidade de mutação
* Probabilidade de recombinação
* Tamanho de torneio
* Elitismo

O algoritmo funciona do seguinte modo:

Inicialização de parâmetros

Enquanto não é atingida a condição de termino

Cria uma nova população com estes passos:

* + - Elitismo
    - Torneio
    - Recombinação com um corte
    - Mutação

Calcula fitness e ordena a nova população

Mantendo os melhores genes da população inicial, preenche os restantes desta mesma com os melhores da nova população

Calcula fitness e ordena a nova população original

Fim do ciclo

De seguida estão explicados os passos do algoritmo:

4.4.2.1 Elitismo

O elitismo é um parâmetro de entrada que age como uma forma de seleção. Aplicamos o elitismo à população recebida no inicio de cada iteração do algoritmo de modo a manter os seus melhores indivíduos (no nosso caso temos a população organizada de melhor para pior). Ou seja, tendo uma população de 100 indivíduos e um elitismo de 20%, irão ser mantidos os primeiros 20 indivíduos.

Os primeiros x indivíduos resultantes da aplicação do elitismo à população inicial, passam a ser os primeiros x indivíduos da nova população.

4.4.2.2 Torneio

Em cada torneio, é escolhido aleatoriamente um elemento do conjunto de indivíduos inicial, este que irá ser utilizado como "melhor individuo".

Para cada *n* iterações (sendo *n* o tamanho do torneio) é escolhido um outro individuo aleatoriamente do mesmo conjunto inicial, para que o seu fitness seja comparado com o do atual melhor individuo. Caso o fitness deste segundo individuo seja melhor, então este toma o lugar do melhor individuo.

No fim do torneio, é devolvido o melhor individuo encontrado.

Nesta fase, teremos já alguns indivíduos na nova população graças ao elitismo, e através do torneio, serão atribuídos os restantes indivíduos a esta população.

4.4.2.3 Recombinação com um corte

Primeiro, iremos percorrer todos os indivíduos da nova população (saltando 1 com cada iteração, sendo que a recombinação será com o próximo vizinho). Em cada iteração começamos por testar se iremos recombinar (ou não) com base no parâmetro de probabilidade de recombinação, de seguida, determinamos aleatoriamente o corte a partir do número de elementos do individuo, por fim, iteramos o valor do corte e substituímos cada elemento com o elemento na mesma posição pertencente ao próximo individuo.

Ao concluir este processo teremos a nova população recombinada.

4.4.2.4 Mutação

Primeiro, iremos percorrer cada individuo da nova população. Com cada iteração, percorremos todos os elementos de cada um desses indivíduos. Para cada elemento, testamos a probabilidade de mutação para saber se iremos mutar (ou não). Caso seja para mutar trocamos o valor desse elemento (se for 1 passa a 0, se for 0 passa a 1).

Ao concluir este processo, teremos a nova população mutada.

* + 1. Bee Colony Optimization

O BCO é baseado na forma como as abelhas…

O algoritmo funciona da seguinte maneira:

Inicialização de parâmetros

Por cada geração

Cálculo das probabilidades

Por cada abelha selecionada

Calcular o número de iterações com base na probabilidade

Por cada iteração

Otimizar solução

FIM

FIM

Calcula fitness e ordena a nova população

FIM

De seguida estão explicados os passos do algoritmo:

4.4.3.1 Inicialização de parâmetros

Nesta etapa iniciamos os parâmetros: número de abelhas selecionadas (selectedSize), número de abelhas consideradas melhores (bestSize), valor base de iterações para as abelhas selecionadas (valueSelection), valor base de iterações para abelhas consideradas melhores (valueBest), número de modificações (changeValue).

4.4.3.2 Inicialização de probabilidades

A cada abelha selecionada e considerada melhor, é atribuído um valor entre 0 e 1 com base na distância do seu fitness ao melhor fitness da população.

A cada iteração este valor de probabilidade é multiplicado pelo número base de iterações fornecido nos parâmetros, para obter o número de iterações em que cada individuo é otimizado numa geração.

4.4.3.3 Otimizar solução

Otimizar solução consiste em para o número de modificações, remover ou adicionar um regenerador até o fitness do individuo melhorar, se o fitness não melhorar o individuo não é alterado.

A probabilidade de remover um regenerador é o dobro da probabilidade de adicionar um regenerador.

A otimização pode ser feita de duas formas, considerar qualquer regenerador livre ou ocupado como alvo de otimização, ou à medida que o número de modificações aumenta, o mesmo numero de melhores e piores regeneradores são considerados para ser modificados (sendo considerados como melhores regeneradores os que estão desligados e podem ligam mais nós, e como piores os que estão ligados e ligam menos nós).

* + 1. Ant Colony Optimization

O ACO é baseado na forma como as formigas comunicam e se influenciam entre si através de trilhos de feromonas. Requer uma população inicial e vários parâmetros que serão referidos de seguida. O elemento mais importante é a variável “trilho de feromonas” que contem toda a base do sistema, pois irá influenciar as outras formigas e sendo atualizado conforme a experiência do algoritmo. Acrescentamos também, que foi utilizado um método de Busca Local de modo a melhorar as soluções encontradas.

O algoritmo funciona da seguinte maneira:

Inicialização de parâmetros

Função de Avaliação

Inicialização do trilho de feromonas

Por cada geração

Por cada formiga na população

Clonar a atual formiga para um individuo auxiliar

Por cada modificação pretendida

Testa probabilidade exploração

Testa intensificação

FIM

Busca local

Função de Avaliação

Atualiza o trilho de feromonas

FIM

De seguida estão explicados os passos do algoritmo:

4.4.4.1 Inicialização de parâmetros

Nesta etapa iniciamos os parâmetros: valor que inicia o trilho de feromonas (Q), probabilidade de exploração (probability\_q), número de modificações (number\_mods), evaporação do trilho de feromonas (evaporation) e influencia do trilho de feromonas (influence).

Iniciamos também o trilho de feromonas em si, que é um vetor de 3 dimensões.

4.4.4.2 Função de Avaliação

Primeiro, é calculado o fitness de cada formiga, de modo a encontrar a melhor (que é depois guardada como a melhor formiga da geração atual). Caso seja a primeira geração ou caso o fitness da melhor formiga da geração seja melhor que o fitness da melhor formiga já encontrada ao longo das várias gerações, então, guardamos a melhor formiga, a melhor geração e ativamos a intensificação.

4.4.4.3 Inicialização do trilho de feromonas

Aqui estabelecemos os valores do trilho de feromonas. O valor atribuído para cada trilho é o mesmo.

T = 1/(Q \* Fitness(G))

Sendo G o melhor individuo encontrado.

4.4.4.4 Clonar a atual formiga para um individuo auxiliar

Desta forma, modificamos o clone da formiga em vez da formiga em si, testando melhoramentos sem consequências.

4.4.4.5 Testa probabilidade exploração

O algoritmo faz o determinado numero (indicado pelo utilizador) de modificações. Para cada uma destas, começamos por testar a probabilidade q. Independentemente do valor obtido, a formiga atual irá sempre interagir com o trilho de feromonas.

4.4.4.6 Testa intensificação

A intensificação é um parâmetro que é desativado quando não foi encontrada uma melhor solução naquela geração, e ativado quando é encontrada uma melhor solução. Isto permite uma exploração mais complexa. Apos testar a intensificação, caso o fitness do individuo auxiliar for melhor que o do melhor correspondente nas formigas, então o individuo auxiliar vai substituir o que está nas formigas.

4.4.4.7 Busca Local

De modo a melhorar os nossos resultados, implementamos uma busca local. Essencialmente, tentamos trocar o valor de um dos elementos de uma formiga e verificamos se melhora o fitness da solução.

4.4.4.8 Atualiza o trilho de feromonas

Os trilhos de feromonas são atualizados com as seguintes equações:

T = (1 - evaporação) \* T

T = T + (influencia/fitness(G))

Sendo G a melhor solução encontrada.

4.4.4.9 Notas

Inicialmente, foi aplicado um sistema que contava as gerações passadas sem melhorias. Não é usado agora, dada a importância de ter o mínimo de regeneradores.

* 1. Síntese

1. - Resultados

Este capítulo está estruturado da seguinte forma: A secção 5.1 apresenta a forma como foram obtidos dados para analisar; A secção 5.2 apresenta a forma como foi feito o tratamento dos dados obtidos, e a secção 5.3 sumariza os conteúdos apresentados neste capítulo.

* 1. Obtenção de dados

Esta secção apresenta as etapas porque passamos de forma a obter os resultados prontos a ser tratados. Iremos explicar como obtivemos os dados e os valores ideais através de testes e do software RapidMiner.

* + 1. Teste de parâmetros dos algoritmos

Para cada algoritmo foram testadas diferentes combinações de parâmetros para o funcionamento do algoritmo, estas diferentes combinações foram aplicadas a 36 problemas, e os resultados da execução foram escritos para ficheiros csv. Os dados escritos nos ficheiros consistem no valor dos parâmetros escolhidos, o tempo que o algoritmo demorou a obter a melhor solução, a geração em que o algoritmo chegou à melhor solução, o número de regeneradores da melhor solução, e o número de nós desligados na melhor solução. Foi também criado um ficheiro de texto com os parâmetros do algoritmo que são constantes a todas as execuções dos testes do algoritmo.

* + 1. Recolha de dados de execução dos algoritmos

Para cada algoritmo foram executados 480 diferentes instâncias de problemas, cada uma com 50 seeds diferentes, os resultados destas execuções forma escritos para ficheiros csv. Os dados escritos nos ficheiros consistem no tamanho do problema, o número do problema, a instância do problema, o tempo que o algoritmo demorou a obter a melhor solução, a geração em que o algoritmo chegou à melhor solução, o fitness da melhor solução, o número de regeneradores da melhor solução, e o número de nós desligados na melhor solução. Foram também criados ficheiros de texto com os parâmetros utilizados na execução dos algoritmos.

* 1. Tratamento dos dados obtidos

Os dados obtidos foram importados para a ferramenta RapidMiner, onde foram aplicadas funções de agregação de forma a condensar milhares de linhas de dados em poucas linhas de informação relevante.

* + 1. Tratamento dos dados de teste de parâmetros

Após a importação dos dados para a ferramenta RapidMiner, foi aplicada a função de agregação, em que os dados são agrupados pelos diferentes atributos que foram variados durante os testes, e são calculadas as médias dos atributos, tempo, gerações, regeneradores, e nós desligados. As médias obtidas são ordenadas por nós desligados de forma crescente, número de regeneradores de forma crescente, e tempo de forma crescente.

A primeira linha do *data set* obtido contem a melhor combinação de atributos para o algoritmo testado.

* + 1. Tratamento dos dados de execução dos algoritmos

Após a importação dos dados para a ferramenta RapidMiner, foi aplicada a função de agregação, em que os dados são agrupados pelos atributos número do problema e tamanho do problema, são filtradas as entradas em que o problema não ficou resolvido (entradas com nós desligados), e são calculadas as médias dos atributos, tempo, gerações, fitness, regeneradores, e nós desligados.

Assim os dados dos diferentes seeds e das diferentes instâncias de cada problema ficam agrupados, de forma a obter um sumario da execução do algoritmo para cada um dos problemas.

* 1. Síntese

Com a ferramenta RapidMiner foi possível concentrar a informação obtida da execução de problemas no programa desenvolvido, de forma a ser mais fácil tirar conclusões dos dados obtidos.

1. - Conclusão

Este capítulo está estruturado da seguinte forma: A secção 6.1 apresenta a comparação dos dados obtidos para os diferentes algoritmos; A secção 6.2 apresenta a escolha dos melhores algoritmos; A secção 6.3 apresenta o trabalho a ser desenvolvido no futuro, e a secção 6.4 sumariza os conteúdos apresentados neste capítulo.

* 1. Comparação dos algoritmos implementados
  2. Escolha dos melhores algoritmos
  3. Trabalho futuro

Devido ao sucesso deste trabalho, surge a possibilidade da sua continuação. Um possível futuro projeto seria a criação de uma aplicação que mapeasse redes de fibra ótica do mundo real para o formato aceite pela nossa aplicação, deste modo possibilitando a utilização do nosso programa para problemas do mundo real. Poderíamos também evoluir a nossa aplicação para que consiga resolver mais problemas, em menos tempo, com mais algoritmos, com a obtenção do hardware necessário.

O ideal uso da nossa aplicação seria mesmo no planeamento da rede de fibra ótica, antes de esta ser instalada. Ou seja, o engenheiro responsável por determinar o seu mapeamento no mundo real, lê o seu planeamento com o nosso software. Deste modo poderá estimar de forma realista os custos, podendo evitar gastos inesperados resultantes de queixas por falta de qualidade por parte de clientes do serviço.

* 1. Síntese

# Bibliografia

|  |  |
| --- | --- |
|  | [Vienna University Computer Center - RLP](http://homepage.univie.ac.at/ivana.ljubic/research/rlp/)  [Qt Documentation](http://doc.qt.io/qt-5/index.html) |

x