

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.

A aplicação está feita de forma a permitir que seja resolvido um problema individualmente, com um gráfico da evolução do fitness do melhor individuo ao longo das gerações, permite também resolver um conjunto de problemas com os mesmos parâmetros e 50 seeds diferentes e escrever para ficheiros csv os resultados, e permite testar diferentes combinações de parâmetros para o mesmo algoritmo para um conjunto de problema.

Os resultados são analisados com auxílio do programa “Rapid Miner”, de forma a conseguir obter as medias de fitness, regeneradores, nós desconectados, e tempo decorrido, para os diferentes seeds e instancias dos problemas.

Abstract

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 (RLP - Regenerator Location Problem) procura que o subconjunto de regeneradores seja instalado a um custo mínimo, de modo a que cada par de nós possa comunicar entre si.

Índice

[Agradecimentos iii](#_Toc516586408)

[Resumo v](#_Toc516586409)

[Abstract vii](#_Toc516586410)

[Índice ix](#_Toc516586411)

[Lista de Figuras xi](#_Toc516586412)

[Lista de Tabelas xiii](#_Toc516586413)

[Lista de Acrónimos xv](#_Toc516586414)

[Capítulo 1 1](#_Toc516586415)

[Estrutura do Programa 1](#_Toc516586416)

[1.1 Linguagem e Framework 1](#_Toc516586417)

[1.1.1 C++/CLI 1](#_Toc516586418)

[1.1.2 Qt 2](#_Toc516586419)

[1.2 Classes 3](#_Toc516586420)

[1.2.1 Algorithm 5](#_Toc516586421)

[1.2.2 Dialog 5](#_Toc516586422)

[1.2.3 MultiThread 6](#_Toc516586423)

[1.2.4 TestDialog 7](#_Toc516586424)

[1.2.5 TestMultiThread 7](#_Toc516586425)

[1.2.6 Thread 8](#_Toc516586426)

[1.2.7 Individual 8](#_Toc516586427)

[1.2.8 MainWindow 9](#_Toc516586428)

[1.2.9 Population 10](#_Toc516586429)

[1.2.10 Problem 10](#_Toc516586430)

[1.3 Síntese 11](#_Toc516586431)

[Capítulo 2 13](#_Toc516586432)

[Algoritmos Implementados 13](#_Toc516586433)

[2.1 Secção …. 13](#_Toc516586434)

[2.1.1 Sub-secção 13](#_Toc516586435)

[2.2 Síntese 13](#_Toc516586436)

[Capítulo 3 15](#_Toc516586437)

[Resultados 15](#_Toc516586438)

[3.1 Obtenção de dados 15](#_Toc516586439)

[3.1.1 Teste de parâmetros dos algoritmos 15](#_Toc516586440)

[3.1.2 Recolha de dados de execução dos algoritmos 15](#_Toc516586441)

[3.2 Tratamento dos dados obtidos 15](#_Toc516586442)

[3.2.1 Tratamento dos dados de teste de parâmetros 15](#_Toc516586443)

[3.2.2 Tratamento dos dados de execução dos algoritmos 15](#_Toc516586444)

[Capítulo 4 18](#_Toc516586445)

[Conclusão 18](#_Toc516586446)

[4.1 Secção …. 18](#_Toc516586447)

[4.1.1 Sub-secção 18](#_Toc516586448)

[4.2 Síntese 18](#_Toc516586449)

[Bibliografia 21](#_Toc516586450)

Lista de Figuras

<criar automaticamente a lista de figuras>

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



Estrutura do Programa

Este capítulo descreve a implementação do programa como um todo, as funcionalidades implementadas e os seus objetivos.

* 1. Linguagem e Framework

Inicialmente o Programa foi implementado em C++/CLI, contudo devido às limitações desta plataforma, acabamos por trocar para a framework de C++ “Qt”.

C++/CLI ou C++ modificado para “Common Language Infrastructure” é a integração de C++ com “Windows Forms”, contudo esta implementação é quase como uma linguagem completamente diferente de C++, sendo que grande parte das instruções não podem ser executadas, e tem também instruções e objetos diferentes.

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. C++/CLI

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

A leitura do problema e feita através da leitura de uma matriz de ligações entre nós, presente em diferentes ficheiros de texto (obtidos pelo site do “Centro de Computadores da Universidade de Viena”), alguns ficheiros têm também pesos associados a cada um dos nós, que representam o custo de colocar um regenerador nesse nó.

A resolução do problema aplica o algoritmo que denominamos de “Custom Algorithm”, os detalhes sobre este estão mais à frente no capítulo 2. Para tal uma nova thread é criada, que vai aplicando o algoritmo e a população vai evoluindo, a cada geração a thread invoca um método do form para o atualizar, no form são mostrados os dados do melhor individuo da população atual, e um gráfico com a evolução do fitness ao longo das iterações do algoritmo.

Figura 1 - Programa em C++/CLI

* + 1. 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 convertido para a classe problema, e de seguida é criada uma thread especifica ao algoritmo selecionado, que a cada geração envia um sinal à thread do form, com a informação do melhor individuo da geração atual, de forma a atualizar os dados na 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 seeds diferentes. São criadas várias threads, cada thread vai resolvendo um problema diferente das restantes, 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, cada thread resolve um problema diferente, e aplica ao mesmo problema as diferentes combinações de parametros, sempre que a solução para um conjunto de parâmetros termina, é enviado um sinal à thread original para escrever os resultados no 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:

setUpAlgorith

generateNewPopulation

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 tambem 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 tambem 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 2 - 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 3 - Ant Colony Test Dialog

* + 1. TestMultiThread

As classes TestMultiThread (AntColonyTestMultiThread, BeeColonyTestMultiThread, CustomTestMultiThread, GeneticTestMultiThread) contêm os mesmos metodos 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 4 - Main Window

* + 1. 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. Síntese

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.



Algoritmos Implementados

Este capítulo descreve os diferentes algoritmos genéticos implementados no projeto.

* 1. Secção ….

Esta secção apresenta … [1].

* + 1. Sub-secção

O protocolo XYZ….

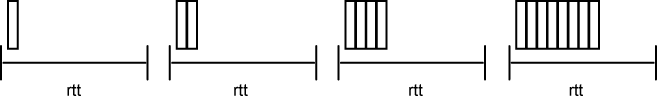


Figura 2.1 - Representação do algoritmo xpto

* 1. Síntese

Este capítulo sintetizou os algoritmos xpto …..



Resultados

Este capítulo descreve a forma como foram obtidos os resultados bem como a forma como estes foram tratados de forma a poderem ser analisados.

* 1. Obtenção de dados

Esta secção apresenta as etapas porque passamos de forma a obter os resultados prontos a ser tratados.

* + 1. Teste de parâmetros dos algoritmos

Para cada algoritmo foram testadas diferentes combinações de dados

* + 1. Recolha de dados de execução dos algoritmos

O

* 1. Tratamento dos dados obtidos

Esta secção apresenta as etapas porque passamos de forma a obter os resultados prontos a ser analisados.

* + 1. Tratamento dos dados de teste de parâmetros

O

* + 1. Tratamento dos dados de execução dos algoritmos

O



Conclusão

Este capítulo descreve.

* 1. Secção ….

Esta secção apresenta … [1].

* + 1. Sub-secção

O protocolo XYZ….

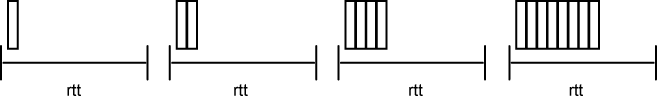


Figura 4.1 - Representação do algoritmo xpto

* 1. Síntese

Este capítulo sintetizou os algoritmos xpto …..

# Bibliografia

x

|  |  |
| --- | --- |
|  | [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