

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 médias de fitness, regeneradores, nós desconectados, e tempo decorrido, para os diferentes seeds e instâncias dos problemas.

REFAZER

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.

REFAZER

Índice

[Agradecimentos iii](#_Toc516756340)

[Resumo v](#_Toc516756341)

[Abstract vii](#_Toc516756342)

[Índice ix](#_Toc516756343)

[Lista de Figuras xi](#_Toc516756344)

[Lista de Tabelas xiii](#_Toc516756345)

[Lista de Acrónimos xv](#_Toc516756346)

[Capítulo 1 – Introdução 1](#_Toc516756347)

[1.1 Breve Enquadramento e Descrição do Problema 1](#_Toc516756348)

[1.1.1 Sub-secção 1](#_Toc516756349)

[1.2 Motivação 1](#_Toc516756350)

[1.2.1 Sub-secção 1](#_Toc516756351)

[1.3 Estrutura do Relatório 1](#_Toc516756352)

[1.3.1 Sub-secção 1](#_Toc516756353)

[1.4 Síntese 1](#_Toc516756354)

[Capítulo 2 - Enquadramento 3](#_Toc516756355)

[2.1 Descrição do Problema 3](#_Toc516756356)

[2.1.1 Sub-secção 3](#_Toc516756357)

[2.2 Problemas de Otimização NP Hard 3](#_Toc516756358)

[2.2.1 Sub-secção 3](#_Toc516756359)

[2.3 Algoritmos evolutivos de inteligência artificial (algoritmos baseados em populações) 3](#_Toc516756360)

[2.3.1 Sub-secção 3](#_Toc516756361)

[2.4 Síntese 3](#_Toc516756362)

[Capítulo 3 - Linguagens e Tecnologias 5](#_Toc516756363)

[3.1 C++ 5](#_Toc516756364)

[3.1.1 C++/CLI 5](#_Toc516756365)

[3.2 Qt 6](#_Toc516756366)

[3.2.1 Qt 6](#_Toc516756367)

[3.3 Github 7](#_Toc516756368)

[3.4 Síntese 7](#_Toc516756369)

[Capítulo 4 - Aplicação 8](#_Toc516756370)

[4.1 Análise de Requisitos 8](#_Toc516756371)

[4.2 Metodologia 8](#_Toc516756372)

[4.3 Estrutura do Programa 8](#_Toc516756373)

[4.3.1 Classes 8](#_Toc516756374)

[4.3.1.1 Algorithm 9](#_Toc516756375)

[4.3.1.2 Dialog 10](#_Toc516756376)

[4.3.1.3 MultiThread 11](#_Toc516756377)

[4.3.1.4 TestDialog 11](#_Toc516756378)

[4.3.1.5 TestMultiThread 12](#_Toc516756379)

[4.3.1.6 Thread 12](#_Toc516756380)

[4.3.1.7 Individual 13](#_Toc516756381)

[4.3.1.8 MainWindow 14](#_Toc516756382)

[4.3.1.9 Population 14](#_Toc516756383)

[4.3.1.10 Problem 14](#_Toc516756384)

[4.4 Algoritmos 15](#_Toc516756385)

[4.4.1 Algoritmo Genético 15](#_Toc516756386)

[4.4.1.1 Descrição do Algoritmo 16](#_Toc516756387)

[4.5 Síntese 16](#_Toc516756388)

[Capítulo 5 – Resultados 17](#_Toc516756389)

[5.1 Obtenção de dados 17](#_Toc516756390)

[5.1.1 Teste de parâmetros dos algoritmos 17](#_Toc516756391)

[5.1.2 Recolha de dados de execução dos algoritmos 17](#_Toc516756392)

[5.2 Tratamento dos dados obtidos 17](#_Toc516756393)

[5.2.1 Tratamento dos dados de teste de parâmetros 17](#_Toc516756394)

[5.2.2 Tratamento dos dados de execução dos algoritmos 17](#_Toc516756395)

[5.3 Síntese 17](#_Toc516756396)

[Bibliografia 19](#_Toc516756397)

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

1. – Introdução

Este capítulo…

* 1. Breve Enquadramento e Descrição do Problema

Esta secção apresenta … [1].

* + 1. Sub-secção

Stuff

* 1. Motivação

Esta secção apresenta … [1].

* + 1. Sub-secção

Stuff

* 1. Estrutura do Relatório

Esta secção apresenta … [1].

* + 1. Sub-secção

Stuff

* 1. Síntese

Este capítulo sintetizou os algoritmos xpto …..

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 consiste em garantir que numa rede de fibra ótica os sinais são regenerados de forma a recuperar a qualidade do sinal, que se deteriora com a distância percorrida na rede. Os problemas utilizados no projeto estão presentes no website do centro de computadores da universidade de Viena. Estes problemas são representações simplificadas de redes de fibra ótica, contendo um conjunto de nós interligados. A solução do problema consiste em 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, com a utilização do menor número de regeneradores possível, e 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 que consiga converter uma rede de fibra ótica real numa rede simplificada como as que foram utilizadas.

* + 1. Estrutura do problema

Os problemas utilizados estão contidos no formato de ficheiros de texto. A principal informação contida nestes ficheiros é uma matriz que representa as ligações entre os diferentes nós que compõem o problema. Os ficheiros contêm também o número de nós e o número de ligações do problema. Alguns problemas contêm também um vetor com pesos referente à colocação de regeneradores em cada nó específico.

* 1. Problemas de Otimização NP Hard

NP-difícil (ou NP-hard, ou NP-complexo) na teoria da complexidade computacional, é uma classe de problemas que são, informalmente, "Pelo menos tão difíceis quanto os problemas mais difíceis em NP". Um problema H é NP-difícil se e somente se (sse) existe um problema NP-completo L que é Turing-redutível em tempo polinomial para H (i.e., L?=?TH). Em outras palavras, L pode ser resolvido em tempo polinomial por uma Máquina de Turing não determinística com um oráculo para H. Informalmente, podemos pensar em um algoritmo que pode chamar tal Máquina de Turing Não-Determinística como uma sub-rotina para resolver H, e resolver L em tempo polinomial, se a chamada da sub-rotina leva apenas um passo para computar. Problemas NP-difíceis podem ser de qualquer tipo: problemas de decisão, problemas de pesquisa ou problemas de otimização.

* + 1. Sub-secção

O protocolo XYZ….

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

Algoritmos Evolutivos são algoritmos baseados em uma gama de mecanismos da evolução biológica e serviram para originar conceitos um pouco mais recentes, como o dos Algoritmos Genéticos

A motivação para a construção de tais modelos computacionais surgiu de teorias através das quais a Natureza, por meio de seus recursos, resolveu problemas de complexidade, isto é, determinar quantidade de “recursos” para resolver “problemas”, de sobrevivência. Assim, pode-se dizer que a natureza otimiza seus mecanismos para resolver um ou mais problemas.

A partir de um problema de otimização, mesmo que se desconheça o que se está otimizando, é possível encontrar uma ótima solução, através dos Algoritmos Evolutivos e suas variações. Ou seja, tais algoritmos podem trabalhar em cima de problemas, sem que exista um conhecimento explicito, isto é, que pode tratar-se de paradigmas.

Os Algoritmos Evolutivos buscam tratar estruturas de objetos abstratos de uma população, como, por exemplo, variáveis de um problema de otimização, dos quais são manipulados por operadores inspirados na evolução biológica, que objetivam a busca para a solução de um problema, estes operadores são comumente chamados de operadores genéticos.

* + 1. Sub-secção

O protocolo XYZ….

* 1. Síntese

Este capítulo sintetizou os algoritmos xpto …..

1. - Linguagens e Tecnologias

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++ e à framework “Qt”.

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

Para sincronização do trabalho efetuado foi utilizada a plataforma GitHub com o repositório [RLP\_Qt](https://github.com/xQsme/RLP_Qt) que agora é publico.

* 1. C++/CLI

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.

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

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. 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. Github

Stuff

* 1. Síntese

Stuff

1. - Aplicação

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

* 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 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.

Figura 2 - Desenvolvimento Iterativo e Incremental

* + 1. As Diferentes Etapas de Desenvolvimento

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 o performance do programa foi utilizado o primeiro algoritmo impelementado 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:

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 3 - 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 4 - 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 5 - 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
     1. Algoritmo Genético

Os algoritmos genéticos são inspirados na teoria da seleção natural de Darwin, tratando uma população inicial e sujeitando os seus indivíduos à seleção, mutação e recombinação durante um determinado **número** de iterações (gerações). No fim de todo o processo, o algoritmo irá devolver o melhor individuo encontrado.

4.4.1.1 Descrição do Algoritmo

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 face ao problema. Através do método de seleção 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 população gerada. No final de todo o processo evolutivo o algoritmo devolve o melhor individuo encontrado.

O algoritmo funciona do seguinte modo:

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:

**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%, irei manter 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.

**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.

**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 numero 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.

**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. Síntese

1. – 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

* 1. Síntese

# 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