,

# Universidade do Minho Departamento de Sistemas de Informática

**MIETI, UMINHO 2022/2023**

**Módulo de Gestão Redes**

**Relatório do Projeto de Grupo**

Grupo:



José Gomes

A93XXX

Diogo Cerqueira

A93108

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

Índice

[1. Introdução 3](#_Toc120884615)

[2. Requisitos 4](#_Toc120884616)

[3. Implementação 5](#_Toc120884617)

[3.1 Leitura do circuito 5](#_Toc120884618)

[3.2 Classes 5](#_Toc120884619)

[3.3 Tipos de pesquisa 8](#_Toc120884644)

[4. Teste, Resultados e Discussão 15](#_Toc120884675)

[4.1 Discussão de Resultados 16](#_Toc120884676)

[5. Autoavaliação 17](#_Toc120884677)

[6. Referências 18](#_Toc120884678)

[7. Conclusão 19](#_Toc120884679)

# Introdução

No âmbito da Unidade Curricular de Gestão de Redes foi proposto a realização de um trabalho que tem como objetivo final o desenvolvimento de um projeto de um agente proxy SNMPv2c utilizando o conhecimento adquirido ao longo do semestre.

Este relatório tem comos objetivos principais apresentar uma breve descrição do problema, uma formulação deste, e iremos descrever todas as decisões tomadas por o grupo, e explicar os resultados obtidos.

De forma a realizar o trabalho com sucesso, anteriormente, foi importante relembrar a linguagem de programação Java, cuja foi a linguagem de programação escolhida pelo grupo para a realização deste projeto.

# Requisitos

De forma a garantir o planeamento prévio e adequado da exequibilidade do presente trabalho torna-se essencial identificar claramente os seus requisitos, uma vez que os mesmos servirão de base ao trabalho executado posteriormente.

* 1. **Requisitos funcionais**

Para que o sistema trabalhe com o funcionamento desejado é necessário que cumpra os seguintes requisitos:

* + - Possuir um pacote freeware instalado com suporte a SNMP – o grupo decidiu utilizar o Net-SNMP;
    - Acrecentar mais algum
  1. **Requisitos não funcionais**

Os próximos requisitos, que serão mostrados em baixo são não funcionais o que significa que os mesmo não estão diretamente ligados às funcionalidades do sistema, mas sim relacionados com o tempo de respostas e de fiabilidade. Os requisitos são definidos por:

* Código simples e otimizado, que levará a um menor tempo de resposta do código;
* Código bem comentado para ser fácil a sua compreensão

# Implementação

# 3.1 Linguagem de Programação

De forma a efetuar a realização do código do projeto, foi decidido a utilização da linguagem de programação Java, pois, além de ser a linguagem para a qual os membros do grupo dão preferência, foi possível descobrir através de uma pesquisa que esta possui bibliotecas que facilitam a implementação da primitiva SNMPv2c.

As bibliotecas utilizadas foram:

* **Crypto** – usada para facilitar a implementação da cifra AES que faz a encriptação dos dados comunicados entre o Agent e o Manager
* **Socket** – usada para a criação da socket que permite a comunicação entre o Agent e o Manager
* **LocalDateTime** – usada para funcionalidades que envolvam o tempo
* **Objects** – usada para facilitar a utilização dos objetos do programa

# 3.2 MIB

A nossa MIB é constituída por uma tabela (OperTable) cuja é constituída por variadas entradas. Na figura seguinte é possivel observar a sua constituição:

Diagrama

Descrição gerada automaticamente

**Objetos da MIB:**

**OperTable**:

**OperEntry:**

**idOper:** é o identificador da operação recebida pelo agente. É a chave da tabela.

**typeOper:** mostra o tipo de operação SNMP.

**operArg1:**

**operArg2:**

**idSource:** identificador da fonte do request.

**idDestination:** identificador do destido do request e onde a operação será executada.

**idArg:** OID do objeto da MIB e cujo é o argumentoda operação a ser executada pelo agente.

**valueArg:** valor do objeto referido pelo idArg e cujo é o resultado recebido no agente proxy vindo do agente SNMP.

**typeArg:** tipo de dados do valueArg.

**sizeArg:** tamanho em bytes do valueArg

**ttl:** tempo de vida restante da entrada na tabela. Quando este objeto possuir o valor zero a entrada é retirada e o Manager não poderá aceder aos valores.

**status:**

Este trabalho possui 4 classes:

# Car

# Nesta classe é criado um objeto “Car” que possui os seguintes atributos:

# Size – Integer, é o tamanho do carro

# Velocidade – Integer, é o valor da velocidade instantânea do carro. O valor inicial é 0 e sempre que o carro desloca-se uma posição, este valor é incrementado uma unidade. Já quando o carro sai fora dos limites da pista, este valor volta a 0.

# Posicao – String, valor que indica a posição atual do carro.

# ID – String, é o campo que identifica o carro.

# Aceleracao – Integer, é o valor atual da aceleração. Este pode variar entre 0, -1 e 1.

# Esta classe também é constituída por os tradicionais métodos de set() e get() dos atributos.

# Além disso, possui uma função “AumentaVelocidade()” que incrementa a velocidade e a função “Bateu()” que coloca o valor da velocidade a 0.

# Node

# Nesta classe é originado um objeto “Node” que possui os seguintes atributos:

# Linha – Integer, é o valor inteiro que indica a linha da posição deste nodo

# Coluna – Integer, é o valor inteiro que indica a linha da coluna deste nodo

# Tipo – String, possui o tipo de nodo. Caso este valor seja um ‘X’, este nodo é um limite da pista, caso seja ‘P’ é a posição inicial, entre outros.

# Name – String, é o nome do node, é constituído por o Tipo + Linha + Coluna.

# Peso – Integer, indica o valor do peso associado ao tipo. Como por exemplo, o limite de pista(‘X’) possui um peso de 20, e o caminho (‘-’) possui um peso de 1. Já a partida(‘P’), possui o peso de 0.

# Adicionalmente, esta classe possui as funções set() e get() dos atributos.

# Pista

# Nesta classe é inicializado um objeto denominado “Pista” que compreende os seguintes atributos:

# Nodes – List, é uma lista de todos os nodos.

# Directed – Boolean, indica se o grafo é direcionado. Caso este valor seja “True”, o grafo é direcionado.

# Graph – Dicionário, é um dicionário que armazena os nodos e os seus respetivos vizinhos.

# Heuristica – Dicionário, armazena a heurística associada a cada nodo. Decidimos atribuir a cada nodo um valor de heurística que pode ser obtida pela subtração do número de colunas do circuito pela coluna do nodo. Deste modo conseguimos ter valores de heurística menores para nodos mais a direita.

# Esta é a classe mais densa pois possui todas as funções de procura, as funções de leitura do ficheiro, de criação do grafo, de criação e adição de arestas ao grafo, entre outros.

# 

# Deep-First Search

# Possui como estratégia expandir sempre um dos nós mais profundos do grafo. É uma pesquisa que não é completa, nem ótima.

# 

# 

# 

Fig 1 – Algoritmo de Pesquisa DFS

# Esta pesquisa recebe como argumentos o node inicial, o node final, um array caminho, e um set para colocar os nós visitados. Começamos por adicionar o node inicial ao array caminho e ao set dos visitados. De seguida, caso o node inicial seja igual ao final, este calcula o custo e retorna logo o caminho e o custo.

# Caso sejam diferentes, vamos buscar o vizinho mais à direita. Depois, caso o node adjacente não esteja no set dos nodes já visitados, fazemos a procura e adicionamos a um array com o resultado. Ele efetua isto até chegar à meta.

# Breath-First Search

# A estratégia desta pesquisa é expandir primeiro os nós de menor profundidade. É muito sistemática, é completa e ótima. É mais aconselhável ser usada em problemas de menor dimensão.

# 

# 

Fig 2 – Algoritmo de Pesquisa BFS

# Esta pesquisa recebe como argumentos o node inicial e o final. Depois é criado um set para colocar os nodes já visitados, e uma queue denominada fila. Depois adicionamos o node inicial à fila e aos visitados. De seguida, garantimos que este node não possui nenhum pai.

# Após isto, percorremos os nodes para achar o melhor caminho. Tal como na DFS, caso o node inicial seja igual ao final, este retorna o caminho e o custo do caminho total. Caso sejam diferentes, consultamos os nodes adjacentes, e caso o node adjacente não seja um node do tipo ‘X’ e que não seja já um node visitado, adicionamos à fila e aos visitados e o node atual fica a ser o pai do adjacente. E isto é recebido até que o nó atual seja igual ao node final.

# Pesquisa A estrela (A\*)

# É um tipo de procura informada, utiliza informação sobre o problema para evitar que o algoritmo de procura procure “às cegas”.

# Função que estima o custo da melhor solução que passa por n:

# f(n) = g(n) + h(n)

def A\_star(self, start, end):  
  
 # open\_list is a list of nodes which have been visited, but who's neighbors  
 # haven't all been inspected, starts off with the start node  
 # closed\_list is a list of nodes which have been visited  
 # and who's neighbors have been inspected  
 open\_list = {start}  
 closed\_list = set([])  
  
 # g contains current distances from start\_node to all other nodes  
 # the default value (if it's not found in the map) is +infinity  
 g = {} ## g é apra substiruir pelo peso ???  
  
 g[start] = 0  
  
 # parents contains an adjacency map of all nodes  
 parents = {start: start}  
 n = None  
 while len(open\_list) > 0:  
 # find a node with the lowest value of f() - evaluation function  
 calc\_heurist = {}  
 flag = 0  
 for v in open\_list:  
 if n is None:  
 n = v  
 else:  
 flag = 1  
 calc\_heurist[v] = g[v] + self.getHeuristica(self.getNodeByName(v))  
 if flag == 1:  
 min\_estima = self.calcula\_est(calc\_heurist)  
 n = min\_estima  
 if n is None:  
 print('Path does not exist!')  
 return None  
  
 # if the current node is the stop\_node  
 # then we begin reconstructin the path from it to the start\_node  
 if n == end:  
 reconst\_path = []  
  
 while parents[n] != n:  
 reconst\_path.append(n)  
 n = parents[n]  
  
 reconst\_path.append(start)  
  
 reconst\_path.reverse()  
  
 # print('Path found: {}'.format(reconst\_path))  
 return reconst\_path, self.calcula\_custo\_total(reconst\_path)  
  
 # for all neighbors of the current node do  
 for (m, weight) in self.getVizinhos(n): # definir função getneighbours tem de ter um par nodo peso  
 # if the current node isn't in both open\_list and closed\_list  
 # add it to open\_list and note n as it's parent  
 if m not in open\_list and m not in closed\_list:  
 open\_list.add(m)  
 parents[m] = n  
 g[m] = g[n] + weight  
  
 # otherwise, check if it's quicker to first visit n, then m  
 # and if it is, update parent data and g data  
 # and if the node was in the closed\_list, move it to open\_list  
 else:  
 if g[m] > g[n] + weight:  
 g[m] = g[n] + weight  
 parents[m] = n  
  
 if m in closed\_list:  
 closed\_list.remove(m)  
 open\_list.add(m)  
  
 # remove n from the open\_list, and add it to closed\_list  
 # because all of his neighbors were inspected  
 open\_list.remove(n)  
 closed\_list.add(n)  
  
 print('Path does not exist!')  
  
 return None

# 

Fig 3 – Algoritmo de Pesquisa A\*

# Nesta função, inicialmente, criamos uma lista denominada “open\_list” para adicionar os nodes que já foram visitados, mas que possuem vizinhos. Criámos também uma lista, a “closed\_list” para os nodes que já foram visitados, e cujos vizinhos já foram inspecionados. Também foi criada uma lista que contém as distância do node inicial para os outros nodes.

# Depois, percorremos a “open\_list” e procurámos o node com menor valor da função f(n) indicada na página 10. Em seguida, já com o node encontrado, reconstruimos o caminho desse node até o node inicial e returnamos o caminho e o seu custo. Seguidamente, percorremos os vizinhos do node atual e caso ainda não esteja na open\_list e na closed\_list, este é adicionado e o nó vizinho atual é adicionado aos seus pais.

# Caso já esteja em ambas as listas, avaliamos se é mais rápido visitar primeiro o pai, e depois o filho. Caso seja, atualizamos o node e os seus pais. E caso o node esteja na closed\_list, é passado para a open\_list. No final é retirado o node da open\_list e é passado para a closed\_list.

# Pesquisa Gulosa(Greedy-Search)

# É outro tipo de procura informada, que tal como a A\*, utiliza informação sobre o problema para evitar que o algoritmo de procura procure “às cegas”.

# Função que estima a distância à solução:

# f(n) = h(n)

def greedy(self, start, end):  
 # open\_list é uma lista de nodos visitados, mas com vizinhos  
 # que ainda não foram todos visitados, começa com o start  
 # closed\_list é uma lista de nodos visitados  
 # e todos os seus vizinhos também já o foram  
 open\_list = {start}  
 closed\_list = set([])  
  
 # parents é um dicionário que mantém o antecessor de um nodo  
 # começa com start  
 parents = {start: start}  
  
 while len(open\_list) > 0:  
 n = None  
  
 # encontrar nodo com a menor heuristica  
 for list\_index in open\_list:  
 if list\_index[0] != 'X':  
 if n is None or self.heuristica[self.getNodeByName(list\_index)] < self.heuristica[  
 self.getNodeByName(n)]:  
 n = list\_index  
  
 if n is None:  
 print('Path does not exist!')  
 return None  
  
 # se o nodo corrente é o destino  
 # reconstruir o caminho a partir desse nodo até ao start  
 # seguindo o antecessor  
 if n == end:  
 reconst\_path = []  
  
 while parents[n] != n:  
 reconst\_path.append(n)  
 n = parents[n]  
  
 reconst\_path.append(start)  
  
 reconst\_path.reverse()  
  
 return reconst\_path, self.calcula\_custo\_total(reconst\_path)  
  
 # para todos os vizinhos do nodo corrente  
 for (neighbour, weight) in self.getVizinhos(n):  
 # Se o nodo corrente nao esta na open nem na closed list  
 # adiciona-lo à open\_list e marcar o antecessor  
 if neighbour not in open\_list and neighbour not in closed\_list:  
 open\_list.add(neighbour)  
 parents[neighbour] = n  
  
 # remover n da open\_list e adiciona-lo à closed\_list  
 # porque todos os seus vizinhos foram inspecionados  
 open\_list.remove(n)  
 closed\_list.add(n)  
  
 print('Path does not exist!')  
  
 return None

# 

Fig 4 – Algoritmo de Pesquisa DFS

# Nesta função, inicialmente, criamos uma lista denominada “open\_list” para adicionar os nodes que já foram visitados, mas que possuem vizinhos e adicionamos o nodo inicial do percurso. Criámos também uma lista, a “closed\_list” para os nodes que já foram visitados, e cujos vizinhos já foram inspecionados. Também foi criada um dicionário que contém o antecessor de um nodo

# De seguida, percorremos a open\_list até encontrarmos o nodo com menor heuristica.

# Seguidamente, se esse nodo é o destino, reconstruimos o caminho desde esse nodo até ao nodo inicial seguindo o antecessor. Depois damos reverse() ao array para termos o caminho do nodo inicial até ao final, e retornamos o array do percurso e o custo total do percurso.

# Depois percorremos os vizinhos do nodo atual e caso o vizinho não esteja na open\_list e na closed\_list, adicionámos o vizinho à open\_list e adicionamos o nó atual aos pais desse vizinho. Para finalizar, retirámos o nodo atual da open\_list e passámos para a closed\_list.

* 1. **Descrição das bibliotecas utilizadas**

Para a relaização deste projeto utilizou-se as seguintes bibliotecas:

* + - math
    - numpy
    - networkx
    - matplotlib.pyplot

A biblioteca math permite utilizar funções matemáticas através de funções já pré-definidas. Já a biblioteca numpy facilita o nosso trabalho com os arrays e ajuda-nos a transformar o array numa matriz. Seguidamente, a biblioteca networkx simplica o trabalho com os grafos, pois permite a visualização do grafo. Para finalizar, a biblioteca matplotlib.pyplot visualizar as arestas do gráfico.

# Teste, Resultados e Discussão

Os testes seguintes foram realizados para o ficheiro de texto “Circuito do Mónaco”

**BFS**

Custo = 24

**DFS**

Custo = 26

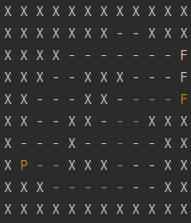
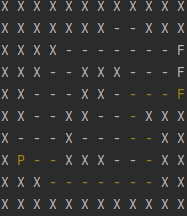


Fig 6 – Percurso da Pesquisa BFS

Fig 5 – Percurso da Pesquisa DFS

**Greedy**

Custo = 26

**A\***

Custo = 24

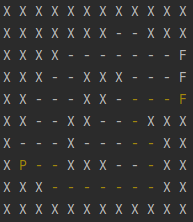
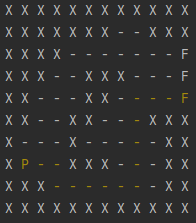
****

Fig 8 – Percurso da Pesquisa A\*

Fig 7 – Percurso da Pesquisa Greedy

# Discussão de Resultados

Como é possível observar pelas figuras 5, 6 , 7 e 8, para este circuito as melhores pesquisas são a BFS e a A\* pois possuem um menor custo final. Também é importante realçar que todas as pesquisas acabaram com uma velocidade final de 17 e que em nenhum momento tiveram uma colisão com os limites de pista.

Através destes resultados, também foi interessante observar que as pesquisas BFS e A\* apesar de possuírem um custo final igual, estas tiveram percursos diferentes.

Também, tal como esperado, conseguimos concluir que a noção geral de melhor ou pior pesquisa não existe, pois essa noção depende do problema proposto. Por exemplo, a pesquisa DFS e Greedy foram as “piores” pesquisas para o Circuito do Mónaco, mas para os circuitos de Melbourne e Suzuki já acabaram com um custo final igual à A\* e à BFS.

# Autoavaliação

* + - Diogo Cerqueira

Neste projeto estive envolvido no desenvolvimento do código e no desenvolvimento do relatório. A maior dificuldade que encontrei durante esta fase foi na elaboração do código da função BFS.

* + - Rui Cunha

Neste projeto estive envolvido no desenvolvimento do código e no desenvolvimento do relatório. A maior dificuldade que encontrei foi na implementação dos códigos de pesquisa.

# Referências

[1] Russell and Norvig (2009). Artificial Intelligence - A Modern Approach, 3rd edition,

ISBN-13: 9780136042594;

[2] Costa E., Simões A., (2008), Inteligência Artificial-Fundamentos e Aplicações, FCA,

ISBN: 978-972-722-34

[3] Maria do Carmo Nicoletti, Estevan R. Hruschka, (2017), Fundamentos da Teoria dos Grafos para Computação

# Conclusão

De modo a finalizar a fase deste relatório, gostávamos de referir que, apesar das dificuldades encontradas consideramos que conseguimos alcançar um bom resultado, pois alcançámos todos os objetivos propostos.

Este trabalho permitiu-nos compreender melhor o funcionamento dos algoritmos de pesquisa e ajudou-nos muito a melhorar a nossa capacidade de trabalhar com a linguagem de programação Python.