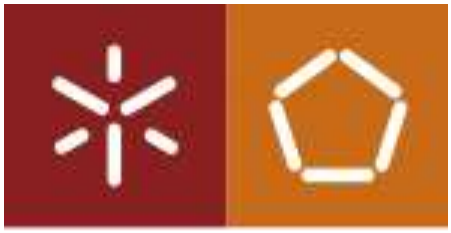

Universidade do Minho
Escola de Engenharia

Green Distribution

Inteligência Artificial

Trabalho em grupo – 2ª Fase

Renato André Machado Gomes	a84696
Sebastião Mendes de Freitas	a71074
Luís Miguel Teixeira Fernandes	a88539
Guilherme Gil Rocha Gonçalves	a88280

05 de janeiro de 2022

1 Introdução

Este documento tem o intuito de apresentar a solução criada face ao problema proposto. Foi deste modo desenvolvido uma continuação do código produzido na fase anterior, de forma a implementar um sistema de recomendação de circuitos de entrega de encomendas.

A solução anterior já dependia de grafos para calcular os caminhos mais curtos, enquanto anteriormente cada estafeta apenas entregava uma encomenda e depois retornava ao ponto inicial, agora estafetas podem fazer várias entregas antes de voltar. Para simular esse comportamento foi criada uma necessidade de calcular o melhor caminho possível, e forçar o caminho a passar por certos pontos do circuito. Como resultado são estabelecidos circuitos e as suas diversas informações, tais como custo total e comprimento. Adicionalmente foi feito um update aos algoritmos de pesquisa usados para gerir os circuitos.

Com a nova implementação, ficou estabelecida uma ferramenta capaz de produzir os cálculos dos circuitos através dos vários algoritmos, possibilitando um estudo dos dados para todos os diversos casos possíveis. Adicionalmente foram criados casos onde são calculados múltiplos circuitos de forma a exigir mais dos algoritmos, assim obtendo dados baseados em amostras maiores.

2 Formulação do problema

O objetivo principal deste projeto, reside no estudo de dados resultantes de procuras, especificamente em viagens onde podem ser feitas entregas a localizações. Foi então criado um grafo baseado em várias localizações (freguesias) do distrito de braga, onde Gualtar ficou a ser o centro de entregas, ou seja, o ponto de partida e o ponto de chegada de qualquer viagem feita por um estafeta.

- Estado inicial: Gualtar;
- Estado objetivo: Entrega de encomendas;
- Operadores: Estafeta;
- Custo da Solução: Comprimento do Ciclo.

Para formular a representação dos dados, foi definido uma viagem feita por um estafeta como um circuito, os quais ficaram a ser as soluções de cada problema de pesquisa requerido. Para chegar a um circuito é primeiro selecionado uma lista de potenciais entregas e um estafeta, onde depois é feita a procura do caminho mais curto através de uma das várias estratégias de pesquisa.

Restrições atribuídas as um potencial circuito:

- Peso total das encomendas;
- Uso do transporte mais ecológico se possível;
- Velocidade do meio do transporte.

Foram usadas cinco estratégias diferentes de pesquisa, onde por teoria, em termos de complexidade estas são os valores esperados.

	Espaço	Tempo
DFS	O(bd)	O(b^d)
BDS	O(b^d)	O(b^d)
Busca iterativa	O(d)	O(b^d)
Gulosa	O(b^m)	O(b^m)
A* (A estrela)	O(b^d)	O(b^d)

b: número médio de arestas ligadas a cada nodo do grafo em pesquisa.
d: profundidade do nodo objetivo da pesquisa.
m: profundidade máxima de pesquisa.

3 Procura não informada

Profundidade (DFS - Depth-First Search):

Conhecido por pesquisar estruturas de dados, onde o algoritmo começa na raiz e explora o máximo possível ao longo de cada ramificação antes de retroceder, em contraste, este é condicionado com a possibilidade de Falhar em ciclos e a apenas apresenta a primeira solução que encontra.

```
resolve_pp_c(Nodo,Destino,Path, C) :-
        profundidadeprimeiro(Nodo,Destino, [Nodo], Caminho, C),
        reverse(Caminho,Path),
        !.

profundidadeprimeiro(Nodo,Destino,Historico, [Destino|Historico], C) :-
        adjacente(Nodo,Destino,C).
profundidadeprimeiro(Nodo,Destino, Historico, Caminho, C) :-
        adjacente(Nodo, ProxNodo, C1),
        ProxNodo \== Destino,
        \+ member(ProxNodo, Historico),
        profundidadeprimeiro(ProxNodo, Destino, [ProxNodo|Historico], Caminho, C2),
        C is C1 + C2.

adjacente(Nodo, ProxNodo, C) :-
        move(Nodo, ProxNodo, C).
adjacente(Nodo, ProxNodo, C) :-
        move(ProxNodo, Nodo, C).

percorreCircuitoProfundidade([],Source, Caminho, Distancia) :-
        resolve_pp_c(Source,gualtar,[H|Caminho],Distancia),
        !.
percorreCircuitoProfundidade([H|L],Source, Circuito, DistanciaTotal) :-
        resolve_pp_c(Source,H,Caminho,Distancia),
        append(Caminho,CircuitoAux,Circuito),
        percorreCircuitoProfundidade(L,H, CircuitoAux, DistanciaAux),
        DistanciaTotal is Distancia + DistanciaAux.
```

O desenvolvimento deste algoritmo baseasse no predicado *profundidadePrimeiro*, o qual recebe dois nodos e calcula o melhor caminho entre os dois. Começa por procurar o primeiro nodo adjacente e se este não é o destino, iria continuar a sua pesquisa, isto é possível através do predicado auxiliar adjacente. O resultado encontrasse invertido, logo para finalizar usamos o predicado *reverse* para obter o caminho em questão, esta operação encontrasse no *resolve_pp_cc*. Finalmente existe ainda a necessidade de criar um caminho que obrigatoriamente percorre certos locais e não apenas vai de um ponto inicial a um ponto final, para tornar isto possível foi criado o predicado *percorrreCircuitoProfundidade*. Assim ficou possível criar um circuito.

Largura (BFS - Breadth-First Search):

Conhecido por pesquisar numa estrutura de dados para um nodo que satisfaça uma determinada propriedade. Ele começa na raiz e explora todos os nós na profundidade atual antes de passar para os nodos no próximo nível de profundidade. Normalmente demora muito tempo e sobretudo ocupa muito espaço

```
percorreCircuitoLargura([],Source, Caminho, Distancia) :-
        larguraPrimeiro([[Source]],gualtar,[H|Caminho],Distancia),
        !.
percorreCircuitoLargura([H|L],Source, Circuito, DistanciaTotal) :-
        larguraPrimeiro([[Source]],H,[C|Caminho],Distancia),
        append(Caminho,CircuitoAux,Circuito),
        percorreCircuitoLargura(L,H, CircuitoAux, DistanciaAux),
        DistanciaTotal is Distancia + DistanciaAux.


percorreL([Node|Path],NewPaths) :-
        findall([NewNode,Node|Path],
                (adjacente(Node,NewNode,_),
                \+ member(NewNode,Path)),
                NewPaths).

larguraPrimeiro(Queue,Goal,Path,Distancia) :-
        larguraPrimeiroAux(Queue,Goal,Caminho),
        reverse(Caminho,Path),
        calculaDistancia(Path,Distancia),
        !.

larguraPrimeiroAux([[Goal|Path]|_],Goal,[Goal|Path]).
larguraPrimeiroAux([Path|Queue],Goal,FinalPath) :-
    percorreL(Path,NewPaths),
    append(Queue,NewPaths,NewQueue),
    larguraPrimeiroAux(NewQueue,Goal,FinalPath).
```

O predicado principal, capaz de calcular um caminho entre dois ponto é o *larguraPrimeiro*, este recorre a um predicado auxiliar *larguraPrimeiroAux* o qual testa se o próximo nodo adjacente está ligado ao ponto final desejado, isto é possível através do predicado auxiliar *percorreL*. Caso não encontre o ponto final recursivamente é testado o próximo nodo adjacente do ponto inicial, se nenhum dos nodos se encontra ligado ao nodo final então aumentamos a profundidade de pesquisa, ou seja, volta ao primeiro nodo testado e corre novamente.

Busca Iterativa Limitada em Profundidade:

Conhecida por combinar a eficiência de espaço da pesquisa em profundidade e a pesquisa rápida da pesquisa em largura (para nodos mais próximos da raiz). A IDDFS chama DFS para diferentes profundidades, começando com um valor inicial. Em todas as chamadas, o DFS é impedido de ir além de uma determinada profundidade. Basicamente, fazemos DFS em um estilo BFS.

```
percorreCircuitoProfundidadeIterativa([],Source, Caminho, Distancia) :-
        iterativa_profundidade([[Source]],gualtar,[H|Caminho],Distancia),
        !.
percorreCircuitoProfundidadeIterativa([H|L],Source, Circuito, DistanciaTotal) :-
        iterativa_profundidade([[Source]],H,[C|Caminho],Distancia),
        append(Caminho,CircuitoAux,Circuito),
        percorreCircuitoProfundidadeIterativa(L,H, CircuitoAux, DistanciaAux),
        DistanciaTotal is Distancia + DistanciaAux.

iterativa_profundidade(Queue,Goal,Path,Distancia) :-
        iterativa_profundidadeAux(1,Queue,Goal,Caminho),
        reverse(Caminho,Path),
        calculaDistancia(Path,Distancia),
        !.

iterativa_profundidadeAux(Depth,Queue,Goal,Path) :-
        profundidadeLimitada(Depth,Queue,Goal,Path).
iterativa_profundidadeAux(Depth,Queue,Goal,Path) :-
        Depth1 is Depth+1,
        iterativa_profundidadeAux(Depth1,Queue,Goal,Path).

profundidadeLimitada(_,[[Goal|Path]|_],Goal,[Goal|Path]).
profundidadeLimitada(Depth,[Path|Queue],Goal,FinalPath) :-
    percorre(Depth,Path,NewPaths),
    append(NewPaths,Queue,NewQueue),
    profundidadeLimitada(Depth,NewQueue,Goal,FinalPath).

percorre(Depth,[Node|Path],NewPaths) :-
    length(Path,Len),
    Len < Depth-1, !,
    findall([NewNode,Node|Path],adjacente(Node,NewNode,_),NewPaths).
percorre(_,_,[]).

calculaDistancia([Aux],Distancia):-
        Distancia is 0,
        !.
calculaDistancia([H,Hs|T],Distancia):-
        Aux = Hs,
        adjacente(H,Hs,C),
        calculaDistancia([Aux|T],DistanciaAux),
        Distancia is DistanciaAux + C.
```

A implementação deste algoritmo baseasse no predicado *iterativa_profundidade*, que recebe dois nodos e o custo caminho entre os dois. Para tal, o predicado *iterativa_profundidaAux*, funciona como o limitador da pesquisa em profundidade, iniciando-a na profundidade 1, o predicado *calculaDistancia* calcula a distancia de um nodo do inicio ao fim do caminho dado pelo predicado *iterativa_profundidaAux*,. No predicado *percorreCircuitoProfundidadeIterativa*, que usando os predicados anteriormente mencionados, devolve um circuito, que inicia e acaba no mesmo local, passando em todos os locais dados em forma de lista.

4 Procura informada

A* (A estrela)

Conhecido por obter a melhor solução possível, esta estratégia de procura tem um grande negativo em termos de utilização de memoria e tempo. Sendo muito mais lento que todos os outros algoritmos estudados neste projeto, é também a melhor solução em termos de resultados. Este algoritmo pesquisa todas as soluções até ao destino e escolhe aquela de menor custo.

```
path(A,B,Path,Len) :-
       travel(A,B,[A],Q,Len),
       reverse(Q,Path).

travel(A,B,P,[B|P],L) :-
       adjacente(A,B,L).
travel(A,B,Visited,Path,L) :-
       adjacente(A,C,D),
       C \== B,
       \+member(C,Visited),
       travel(C,B,[C|Visited],Path,L1),
       L is D+L1.

pesquisaA(A,B,Path,Length) :-
   setof([P,L],path(A,B,P,L),Set),
   Set = [_|_], % fail if empty
   minimal(Set,[Path,Length]).

minimal([F|R],M) :- min(R,F,M).

% minimal path
min([],M,M).
min([[P,L]|R],[_,M],Min) :- L < M, !, min(R,[P,L],Min).
min([_|R],M,Min) :- min(R,M,Min).
```

O predicado principal da pesquisa A* é o *pesquisaA*, usando os predicados auxiliares *path* e *travel*, esta verifica se o nodo atual é adjacente ao nodo destino, se não for verifica se este faz parte de uma lista que contem os nodos já visitados, caso contrário é chamada até o último nodo ser o desejado. O predicado *path* simplesmente organiza através do reverse o caminho dado pelo *travel*. O *pesquisaA* dá uso ao predicado *setOff* que coloca todos os paths dentro de um set, set este que é depois filtrado pelo predicado *minimal* de forma a devolver apenas a solução com menor custo.

```
shortestAll(_, [], C, E,CustoMenor, EncomendaMProx):-
        CustoMenor = C,
        EncomendaMProx = E,
        !.
shortestAll(Source, [H|L], Custo, Encomenda,CustoMenor, EncomendaMProx):-
        pesquisaA(Source,H,_,C1),
        ((C1 >= Custo) -> shortestAll(Source, L, Custo, Encomenda,CustoMenor, EncomendaMProx);
        shortestAll(Source, L, C1, H,CustoMenor, EncomendaMProx)).

percorreCircuito([],_, _, _) :- !.
percorreCircuito([H|L],Source, CustoMenor, EncomendaMProx) :-
        pesquisaA(Source,H,_,Custo),
        shortestAll(Source, L, Custo,H,CustoMenor, EncomendaMProx).

percorreTodoCircuito([],Source, CustoTotal,P) :-
        pesquisaA(Source,gualtar,P,L),
        CustoTotal is L,
        !.
percorreTodoCircuito([H|L],Source, CustoTotal,[Source|Vs]) :-
        percorreCircuito([H|L],Source, CustoMenor, EncomendaMProx),
        remover(EncomendaMProx,[H|L],Lista),
        percorreTodoCircuito(Lista,EncomendaMProx, CustoAux,Vs),
        CustoTotal is CustoAux + CustoMenor,
        !.
```

Existe ainda como nas soluções anteriores a necessidade de fazer uma pesquisa onde é obrigatório passar por certos nodos antes de chegar ao nodo destino, para tornar isto possível foi criado o predicado *percorreTodoCircuito*. O predicado *shortestAll* garante que da lista de locais de entrega, o próximo para qual a viagem ocorre é o mais próximo do nodo atual, garantido assim que a solução seja o mais eficiente possível. O predicado *percorreCircuito* dando uso a *shortestAll* e a *pesquisaA*, seleciona de uma lista de locais o mais próximo, e vai para o nodo em questão. Através do predicado *percorreTodoCircuito*, usando os predicados acima referidos, cria um circuito passando por todos os nodos que lhe são dados, e terminando o circuito em Gualtar.

Gulosa

Conhecido como o algorítmico que constrói uma solução peça por peça, escolhendo sempre a próxima peça que oferece o benefício mais óbvio e imediato.

```
percorreCircuitoGulosa([],Source, Distancia,Circuito) :-
	resolve_gulosa(Source, gualtar, Caminho, Distancia),
	append(Caminho, [gualtar], Circuito),
	!.
percorreCircuitoGulosa([H|L], Source, Distancia, Circuito) :-
	resolve_gulosa(Source, H, Caminho, Custo),
	append(Caminho, CaminhoAux, Circuito),
	percorreCircuitoGulosa(L, H, CustoAux, CaminhoAux),
	Distancia is Custo + CustoAux.

resolve_gulosa(Nodo,Destino,Path, C) :-
	gulosa(Nodo,Destino, [Nodo], [Aux | Caminho], C),
	reverse(Caminho,Path),
	!.

gulosa(Nodo,Destino,Historico, [Destino|Historico], C) :-
	adjacente(Nodo,Destino,C).
gulosa(Nodo,Destino, Historico, Caminho, C) :-
	nodosAdjacentes(Nodo,Next),
	naoMembro(Next, Historico, ProxNodos),
	proxMaisCurto(ProxNodos,ProxNodo,Custo),
	gulosa(ProxNodo, Destino, [ProxNodo|Historico], Caminho, C2),
	C is Custo + C2.

nodosAdjacentes(Source,ProxNodos):-
	findall((C,ProxNodo),adjacente(Source,ProxNodo,C),ProxNodos).

proxMaisCurto(Next,ProxNodo,Custo):-
	msort(Next,[(H,X)|T]),
	ProxNodo = X,
	Custo = H.

naoMembro([], _, []):-!.
naoMembro([(X,H)|L], Historico, ProxNodos) :-
	(member(H,Historico) -> naoMembro(L, Historico, ProxNodos);
	append(ProxNodosAux,[(X,H)],ProxNodos),
	append(Historico,[H],HistoricoAux),
	naoMembro(L, HistoricoAux, ProxNodosAux)).
```

O predicado principal da pesquisa gulosa é o *resolve_gulosa*, este chama o *gulosa* com a ajuda do predicado *nodosAdjacentes*, que cria uma lista com todos os nodos adjacentes ao atual, lista esta que é depois filtrada pelo predicado *naoMembro*, evitando que este volte ao ponto do qual saiu(par que não ocorra um loop infinito). No predicado *proxMaisCurto*, que indica o nodo adjacente mais próximo, chamando recursivamente o *gulosa*, ate chegar ao destino. O predicado *percorreCircuitoGulosa*, recebe uma lista de nodos e usando o *gulosa* conecta-os e regressa ao nodo inicial, formando assim um circuito.

5 Resultados

Caso 1:

Neste primeiro caso de teste vamos olhar para uma viagem bastante longa, ou seja, este vai ser o caso mais complexo possível. Especificamente tratasse de gerir um circuito que força a passagem em todos os nodos do mapa, no total 40 encomendas. Esta caso é irrealista no contexto da Green Destribution, mas foi decidido usar como um caso de estudo para termos acesso a uma perspetiva mais extrema.

Estratégia	Tempo (segundos)	Inferências	Indicador/ custo	Encontrou a melhor solução?
DFS	**0.0000**	**2,941**	**215**	**Não**
BFS	**0.0156**	**14,449**	**173**	**Não**
Iterative deepening	**0.0156**	**6,592**	**173**	**Não**
A *	**0.1720**	**1,579,185**	**69**	**Sim**
Gulosa	**0.0985**	**985,583**	**123**	**Não**

Como esta apresentado é a estratégia A* foi a única que encontrou a solução ótima, a qual é duas vezes mais barata do que qualquer circuito calculado pelos algoritmos de procura não informada e Gulosa. Contudo conseguimos ver através do tempo e das inferências que o algoritmo A* é extremamente pesado em relação as outras estratégias. Já as pesquisas não informadas, como a DFS, BFS, Iterative deepening, apesar de não chegarem a solução ótima e os custos serem elevados, são muito mais rápidas e leves, do as pesquisas informadas.

Caso 2:

Agora passamos para o caso mais simples possível, onde trabalhamos com uma única entrega próxima do ponto incial.

Estratégia	Tempo (segundos)	Inferências	Indicador/ custo	Encontrou a melhor solução?
DFS	**0.0000**	**47**	**6**	**Sim**
BFS	**0.0000**	**304**	**6**	**Sim**
Iterative deepening	**0.0000**	**156**	**6**	**Sim**
A *	**0.0000**	**33,212**	**6**	**Sim**
Gulosa	**0.0000**	**46**	**6**	**Sim**

Em casos como este fica bastante obvio que qualquer algoritmo obtém uma solução ótima, logo podemos potencialmente concluir que em casos simples a estratégia Gulosa é a mais eficiente. Esta caso apenas, não é suficiente para tirar conclusões sobre qual das estratégias deveria ser usada, pois os dados finais são demasiados próximos. Vamos agora num terceiro caso observar um comportamento próximo deste, mas com a diferença de que o local de entrega se encontra o mais longe possível.

Caso 3:

Tal como no caso 2 trabalhamos sobre apenas uma entrega, a diferença é que desta vez a localização pela qual o circuito tem que passar encontrasse o mais longe possível.

Estratégia	Tempo (segundos)	Inferências	Indicador/ custo	Encontrou a melhor solução?
DFS	**0.0000**	**97**	**56**	**Não**
BFS	**0.0000**	**1960**	**56**	**Não**
Iterative deepening	**0.0000**	**829**	**56**	**Não**
A *	**0.0000**	**48,619**	**18**	**Sim**
Gulosa	**0.0000**	**930**	**26**	**Não**

Tal como esperado, o aumento da complexidade do caso em estudo tem um impacto negativo. Agora tal como no caso 1 apenas a procura A* é capaz de encontrar a solução ótima, apesar de uma alta complexidade. Se a eficiência da complexidade do algoritmo é algo bastante desejado, existe um argumento para o uso da Gulosa em vez do A* uma vez que a diferença do custo final não é necessariamente demasiado negativa tendo em conta a vantagem da complexidade do Gulosa, que se torna muito mais simples por um pequeno aumento de custo.

6 Querys

Gera Circuitos

```
geraLocais([],[]):- !.
geraLocais([Enc|T],[H|L]):-
  encomenda(Enc,_,_,Cli,_,_,_,_,_,_,_),
  cliente(Cli,Local),
  H = Local,
  geraLocais(T,L),
  !.

geraCircuitosAEAux(Locais,Distancia,Circuito):-
percorreTodoCircuito(Locais,gualtar, Distancia,Circuito).

geraCircuitosAE(Enc,Distancia,Circuito):-
  geraLocais(Enc,Locais),
  remove_duplicates(Locais,LocaisL),
  geraCircuitosAEAux(LocaisL,Distancia,Circuito).
```

No predicado *geraCircuitosAE*, recebe uma lista de encomendas e traça um circuito de entregas pelas localizações das mesmas, para tal usa o predicado *geraLocais*, que percorre a lista de encomendas e gera uma lista com os locais. E o predicado *geraCircuitosAEAux*, que percorre esses locais utilizando o algoritmo A*.

Criamos também uma versão deste predicado para cada um dos algoritmos implementados.

Entrega mais Rápida

```
entregaMaisRapidaPossivel(Enc,Transporte,Tempo,Distancia):-
  encomenda(Enc,_,_,_,_,_,_,_,_,_,Prazo),
  pesoTotal([Enc],Peso),
  geraCircuitosAE([Enc],Distancia,_),
  escolheTransportePrazo(Peso,Distancia,Prazo,_,TransporteAux),
  (TransporteAux == 'bicicleta' -> Transporte = 'moto', medeVelocidade('moto',Peso,Velocidade), calculaTempoAux(Distancia,Velocidade,Tempo);
    TransporteAux == 'moto' -> Transporte = 'moto', medeVelocidade('moto',Peso,Velocidade), calculaTempoAux(Distancia,Velocidade,Tempo);
      Transporte = 'carro' , medeVelocidade('carro',Peso,Velocidade), calculaTempoAux(Distancia,Velocidade,Tempo)).
```

Este predicado recebe uma encomenda recebe uma encomenda, e diz-nos o transporte mais rápido possível para entregar. Para tal, verifica quais transportes conseguem transportar a encomenda, com base no peso e na prazo dada pelo cliente. Usando então os transportes possíveis, vendo qual seria o mais rápido em termos de tempo.

Entrega mais Ecológica

```
entregaMaisEcologicaPossivel(Enc,Transporte,Tempo,Distancia):-
  encomenda(Enc,_,_,_,_,_,_,_,_,_,Prazo),
  pesoTotal([Enc],Peso),
  geraCircuitosAE([Enc],Distancia,_),
  escolheTransportePrazo(Peso,Distancia,Prazo,Tempo,Transporte).
```

Este predicado recebe também uma encomenda e encontra o transporte mais ecológico possível(bicicleta>mota>carro). Para tal, o predicado *escolheTransportePrazo*, verifica quais os transportes possíveis, verificando o peso da encomenda e calcula também a sua velocidade e tempo para comparar com o prazo dado pelo cliente, filtrando assim os possíveis transportes. A velocidade destes transportes é influenciada pelo peso usando as formulas fornecidas pelos professores.

7 Conclusão

Durante o desenvolvimento desta fase da parte Prática, o grupo foi confrontado com diversos obstáculos, sendo a compreensão completa dos algoritmos com a sua implementação no contexto do projeto a maior dificuldade.

Em contraste, o grupo teve a oportunidade de observar os resultados da utilização dos diversos algoritmos em diferentes casos de estudo, onde após serem analisados, demonstraram que, apesar de obter sempre a melhor solução, a A* é demasiado dispendiosa a nível de memória e de tempo de execução.

Em suma, o grupo teve alguma dificuldade na compreensão do enunciado, mas conseguiu obter resultados que puderam demonstrar as diferenças existentes em relação à maneira com que exercemos a procura sobre um grafo.