Pesquisa aplicada à entrega de encomendas

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Inteligência Artificial

Grupo A4_4:
Luís Barbosa - 201405729 - up201405729@fe.up.pt
Paulo Santos - 201403745 - up201403745@fe.up.pt
Sérgio Ferreira - 201403074 - up201403074@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

21 de Maio de 2017

Conteúdo

1 Objetivo

Este trabalho possui como principal objetivo a pesquisa sistemática, informada ou não informada, de soluções aplicada à entrega de encomendas. Para a pesquisa utilizando métodos fracos, iremos utilizar o algoritmo de pesquisa em profundidade e em largura e para a pesquisa com métodos informados, serão utilizados os algoritmos A* e IDA*.

Neste caso em concreto, a solução que se pretende determinar consiste no melhor percurso a realizar para efetuar uma entrega de um determinado conjunto de encomendas.

Este percurso deve procurar otimizar diferentes critérios, mediante a especificação do utilizador. Pode, por exemplo, procurar maximizar o número de entregas ou, por outro lado, minimizar o espaço percorrido, caso não pretenda que o transporte destas encomendas ocupe um dia completo.

2 Especificação

Tendo em conta que o tema escolhido se trata de um problema de pesquisa, é necessário representar os estados, a função de transição e a heurística.

Para a distribuição de encomendas, numa certa região, são necessários vários camiões. No entanto, neste caso, uma vez que a empresa têxtil em questão possui dois camiões, mas um dos motoristas não compareceu ao trabalho por motivo de doença, apenas se considera que um camião fará a entrega. Os camiões possuem um identificador, uma determinada autonomia fixa (combustível) e uma capacidade máxima de carga (volume). São representados pelo seguinte predicado:

camiao(Id, Autonomia, CargaMaxima).

Num dia de trabalho, o motorista de um camião percorre, em média, 500 quilómetros. É importante ter em conta que um camião nunca deve ficar sem combustível, podendo abastecer em certos pontos do mapa.

Para representar os pontos de um mapa, que irão corresponder aos locais onde serão feitas as entregas ou onde se localizam os postos de abastecimento, implementou-se o seguinte predicado que contém um identificador e uma longitude e uma latitude em graus:

pontoGrafo(Id, Long, Lat).

Para se saber quais os pontos onde se começa e acaba a viagem e os pontos onde se pode efetuar o abastecimento, criaram-se os seguintes predicados:

pontoInicial(Id).
pontoFinal(Id).
pontoAbastecimento(Id).

Em todos os casos, "Id" corresponde ao "Id" de um PontoGrafo.

As encomendas que serão distribuídas são definidas pelo seguinte predicado:

encomenda(Id, Volume, Valor, IdPontoEntrega, Cliente).

Desta forma, cada estado do problema de pesquisa irá ser representado por um PontoGrafo.

A função de transição que consideramos adequada corresponde a um facto que se refere a uma aresta do grafo. Esta aresta será representada por um facto com a seguinte definição em que o custo está em quilómetros:

sucessor(IdPontoGrafo, IdSucessor, Custo).

Para a resolução do problema, serão aplicados alguns algoritmos de pesquisa (pesquisa em profundidade e largura, A* e IDA*), sendo testada a eficácia destes através da medição do tempo de execução que cada um leva a chegar a uma solução (ótima ou não, dependendo do algoritmo) para o problema.

Começando pelos métodos fracos, sabe-se que estes utilizam técnicas genéricas de pesquisa, sendo independentes do problema, podendo levar a uma explosão combinatória do número de estados a pesquisar.

O primeiro algoritmo implementado foi o **"Pesquisa em profundidade"** (DFS), uma vez que é considerado o método de pesquisa mais simples. Este, começando num estado inicial, verifica se é solução. Se for, termina, senão, efetua *backtrack*. Se o nó não possuir sucessores, faz-se *backtrack*. De seguida, escolhe-se um filho do nó ainda não visitado, repetindo-se o algoritmo a partir desse filho. Se existirem nós por visitar, volta-se a escolher um que ainda não tenha sido visitado, senão faz-se *backtrack*.

```
pp(Ei,Ef,Custo,Caminho) :-
  pp(Ei,Ef,[Ei],L,Custo),
  reverse(L,Caminho).
```

```
pp(Ef,Ef,L,L,0).
pp(Ea,Ef,Lant,L,Custo) :-
   sucessor(Ea,Eseg,C),
   \+ member(Eseg,Lant),
   pp(Eseg,Ef,[Eseg|Lant],L,Custo2),
   Custo is Custo2 + C.
```

Seja:
r - fator de ramificação média;
p - profundidade máxima;
O algoritmo, para cada ramo, terá de o percorrer até à sua profundidade máxima (p), ou seja, terá complexidade temporal O(r^p).

Por outro lado, apenas necessita de armazenar o estado dos filhos dos nós que fazem parte da solução atual, ou seja, a sua complexidade espacial será apenas O($r * p$).

Não é considerado um algoritmo completo, uma vez que poderia nunca chegar a encontrar uma solução caso o caminho fosse infinitamente grande.

Também não é ótimo, visto que a primeira solução que encontra pode não ser a que necessita de menos passos.

De seguida, e ainda nos métodos fracos, decidimos implementar a **"Pesquisa em largura"**. Neste algoritmo, é dada prioridade aos nós mais próximos da raiz. O algoritmo, começando na raiz e com uma fila vazia, verifica se o nó inicial é solução. Se for, termina o algoritmo, tendo uma solução garantidamente ótima. Se o nó possuir sucessores, estes são adicionados ao fim da fila. Retira-se o nó que estiver no topo da fila e repete-se o algoritmo para esse nó.

```
custoTotal([_],0).
custoTotal([P1,P2],Custo) :-
   sucessor(P1,P2,Custo).
custoTotal([P1,P2|Ps],CustoTotal) :-
   sucessor(P1,P2,Custo),
   custoTotal([P2|Ps],CustoTotal2),
   CustoTotal is Custo + CustoTotal2.

pl(Ei,Ef,Custo,Caminho) :-
   pl([[Ei]],Ef,L),
   reverse(L,Caminho),
   custoTotal(Caminho,Custo).

pl([La|_],Ef,La) :- La=[Ef|_].
pl([La|OLs],Ef,L) :-
   La=[Ea|OEs],
   findall([Eseg|La],(sucessor(Ea,Eseg,_),
   \+ member(Eseg,OEs)),Lseg),
   append(OLs,Lseg,NL),
   pl(NL,Ef,L).
```

Este algoritmo apenas visita a árvore até à profundidade da solução, fazendo com que a complexidade temporal seja O(r^p), assim como a complexidade espacial, uma vez que a fila vai ter de armazenar todos os nós do nível da solução.

Se todas as arestas possuíssem o mesmo custo, o algoritmo seria considerado ótimo, pois a primeira solução encontrada seria necessariamente a melhor, apesar de não ser o caso. No entanto, considera-se que é um algoritmo completo, uma vez que r é finito.

Comparando estes dois algoritmos, verifica-se que o **"Primeiro em profundidade"** utiliza menos memória que o **"Primeiro em largura"**.

Uma vez que os métodos fracos não possuem informação sobre o problema, decidimos implementar dois métodos informados, pois estes possuem alguma, nomeadamente algum tipo de função heurística que permita dar um valor a um determinado estado. Com este valor, é possível escolher aquela que se acredita ser a melhor opção, ignorando as outras.

Uma vez que uma função heurística não deve ser uma função qualquer, tentamos que a escolhida tivesse as seguintes propriedades:

- Mais simples que o cálculo do valor real do estado, sendo, no entanto, o mais próxima do valor real possível (pois desta forma, mais depressa o método converge para a solução);

- Ser otimista. Como no nosso caso, pretendemos minimizar o custo, a função heurística tem que ser menor ou igual ao custo real;

- Ser consistente (monótona).

Uma vez que o nosso problema se trata de distâncias físicas entre pontos, consideramos que a função heurística mais adequada seria a distância em linha reta entre os 2 pontos. Esta heurística é claramente otimista, uma vez que não é possível a distância real ser menor que a nossa função. A heurística calcula a distância em graus entre dois pontos na Terra e converte essa distância para quilómetros. Para simplificar, não considera arcos entre dois pontos, mas sim rectas.

```
heuristica(IdPontoGrafo,Hseg) :-
  pontoFinal(IdPontoFinal),
  pontoGrafo(IdPontoFinal,Long1,Lat1),
  pontoGrafo(IdPontoGrafo,Long2,Lat2),
  Long is abs(Long2 - Long1),
  Lat is abs(Lat2 - Lat1),
  Hipotenusa is sqrt(Long * Long + Lat * Lat),
  Hseg is 40000 * Hipotenusa / 360.
```

Após estar definida a heurística, implementamos o algoritmo **A***. Este considera que o valor de cada sucessor é dado pela seguinte função:

f*(n) = g(n) + h*(n).

Esta função resulta de uma combinação dos algoritmos gananciosos (h*(n) corresponde à estimativa do custo do passo desde o nó corrente até à solução), que realizam estimativas do futuro, e do algorimo do custo uniforme, que utiliza os custos até ao presente (g(n)).

Este algoritmo recorre a duas listas:

- Lista aberta: Nós a expandir, ordenados por f*(n).

- Lista fechada: Nós já expandidos.

Inicialmente, a lista aberta contém apenas o estado inicial e a lista fechada está vazia. Caso a lista aberta esteja vazia, o algoritmo termina. Se não, retira-se o primeiro estado da lista aberta e adiciona-se este à lista fechada. Se o estado for o estado final, o algoritmo retorna. Caso contrário, para cada sucessor que não esteja na lista fechada, calcula-se o seu possível novo valor estimado e se o sucessor não estiver na lista aberta ou o novo valor for melhor que o valor anterior, o sucessor é adicionado à lista aberta. Repete-se o algoritmo até este terminar.

```
astar(PontoInicial,PontoFinal,Caminho,Custo) :-
  heuristica(PontoInicial,Hi),
  astar(PontoInicial, PontoFinal,[Hi-[PontoInicial]-0],L,Custo),
```

```
  reverse(L,Caminho).

astar(_PontoInicial,E,[C-[E|Cam]-_|_],[E|Cam],C) :- !.
astar(PontoInicial,PontoFinal,[_-[E|Cam]-G|R],S,C) :-
  findall(F2-[E2|[E|Cam]]-G2,
    (sucessor(E,E2,C),G2 is G + C,heuristica(E2,H2),F2 is G2 + H2),
    Lsuc),
  append(R,Lsuc,L),
  sort(L,Lord),
  astar(PontoInicial,PontoFinal,Lord,S,C).
```

Apesar da complexidade do algoritmo A* depender muito da função heurística escolhida, pode-se afirmar que em todos os casos este algoritmo é bastante dispendioso na memória, uma vez que tem que armazenar a informação de todos os estados até chegar ao estado final, e também gasta bastante tempo.

Por não se aplicar a problemas de larga escala, achamos importante implementar o algoritmo **IDA*** com aprofundamento iterativo para reduzir a memória necessária, utilizando-se o valor da função heurística em vez da profundidade para parar e recomeçar o algoritmo.

```
idastar(Ei,Ef,Custo,Caminho) :-
  retract(next_bound(_)),
  fail
  ;
  asserta(next_bound(0)),
  idastarAux(Ei,Ef,L),
  reverse(L,Caminho),
  custoTotal(Caminho,Custo).

idastarAux(Ei,Ef,L) :-
  retract(next_bound(Bound)),
  asserta(next_bound(100000)),
  heuristica(Ei,Hi),
  pp1([Ei],Ef,Hi,Bound,L)
  ;
  next_bound(NextBound),
  NextBound < 100000,
  idastarAux(Ei,Ef,L).

pp1([E|OEs],E,H,Bound,[E|OEs]) :- H =< Bound.
pp1([E|OEs],Ef,H,Bound,Sol) :-
  H =< Bound,
  sucessor(E,Esuc,_),
  \+ member(Esuc,OEs),
  heuristica(Esuc,Hsuc),
  pp1([Esuc,E|OEs],Ef,Hsuc,Bound,Sol).
pp1(_,_,H,Bound,_) :-
  H > Bound,
  update_next_bound(H),
  fail.

update_next_bound(H) :-
  next_bound(Bound),
```

```
Bound =< H, !
;
retract(next_bound(Bound)), !,
asserta(next_bound(H)).
```

3 Desenvolvimento

3.1 Ferramentas/APIs utilizadas

Para o desenvolvimento de uma solução para o problema apresentado, o grupo decidiu que seria uma boa ideia tirar partido das funcionalidades da linguagem **Prolog** para implementar tanto os algoritmos como os predicados que solucionam o enunciado em questão. Este código foi desenvolvido em **Notepad++**, fazendo uso de um ficheiro que continha a síntaxe da linguagem **Prolog** para o IDE em questão, de forma a fornecer uma melhor legibilidade do código. Para compilar e correr o programa, utilizou-se **SICStus Prolog VC14 4.3.3**.

Além da solução em si, e de forma a facilitar o teste desta, o grupo implementou um programa gerador de dados que construísse vários factos com informações pertinentes para o problema. Este programa foi implementado em **C++**, fazendo-se uso do **Visual Studio 2017**.

Todo o ambiente de desenvolvimento deu-se em **Windows 10**.

3.2 Estrutura da aplicação

A aplicação referente à solução do problema foi implementada num único ficheiro **Prolog**, denominado **"Entrega de encomendas.pl"**. Inicialmente, é carregado um ficheiro com factos denominado **"data1.pl"**, que foi criado usando o programa de geração de dados. De seguida, encontram-se implementados os quatro algoritmos referidos anteriormente, que são utilizados pelos predicados que resolvem o problema, estando estes implementados a seguir aos algoritmos.

Uma vez que se trata de uma implementação em **Prolog**, consideramos que não faz sentido uma descrição por módulos ou mesmo a demonstração de diagrama de classes.

No entando, o programa de geração de dados encontra-se dividido em dois ficheiros: "DataGenerator.cpp"e "Node.cpp"(com o respetivo .h associado). Este programa é bastante simples: pede ao utilizador o directório de um ficheiro com nomes de clientes, pede o nome do novo ficheiro para onde os dados devem ser enviados e pede um conjunto de informações sobre quantos dados deve gerar e como os deve gerar. Com base nestas informações vai gerar dados aleatórios entre certos intervalos e assegurar que não são gerados factos repetidos. O ficheiro gerado pode ser indicado no ficheiro "Entrega de encomendas.pl"e, assim, ser usado pela aplicação.

O ficheiro "DataGenerator.cpp"contém toda a lógica do gerador; os ficheiros "Node.cpp"e "Node.h"apenas servem para que se possa guardar os dados de um nó num objecto e que este, ao ser guardado num set, seja único nesse set, dadas as propriedades da classe set.

3.3 Detalhes relevantes da implementação

Passando para a descrição da implementação da solução, esta é realizada através da chamada ao seguinte predicado:

entregaEncomendas(Algoritmo,Opcao).

Em Algoritmo, pode-se colocar as seguintes opções: **pp, pl, astar** ou **idastar**. Em Opcao, pode-se colocar **maxEntregas** caso se pretenda determinar o percurso de forma a maximizar o número de entregas, ou **minDist** caso se queira minimizar o espaço percorrido pelo camião.

Inicialmente, é chamado o seguinte predicado:

todasEncomendas(EncomendasTemp,Opcao,Algoritmo)

ao qual se passam os dados anteriores e onde se obtém, em Encomendas-Temp, todas as encomendas que um camião poderá vir a suportar, dada a sua capacidade máxima. Neste predicado, começa-se por obter todas as encomendas que estão na nossa base de factos, sendo colocadas numa lista no formato Volume-PontoGrafo. Caso a opção escolhida seja maximizar as entregas, a lista é ordenada da encomenda que possui menor volume para a maior, garantindo-se assim que no camião se colocam o máximo de encomendas possíveis. Caso a opção seja minimizar a distância, chama-se um predicado com o seguinte formato: **ordenaEncomendasDistancia(Ei,Encomendas,Algoritmo,Sorted).** Neste predicado, passa-se o ponto inicial do nosso grafo, todas as encomendas que foram calculadas inicialmente e o algoritmo em questão. Mediante o algoritmo, é colocado em Sorted no formato Volume-PontoGrafo a encomenda mais próxima do ponto inicial e, de seguida, o predicado é chamado recursivamente para calcular qual é a próxima encomenda mais próxima da anterior, e assim sucessivamente, até a lista que continha todas as encomendas se encontrar vazia. Desta forma, garante-se a escolha de um caminho mais curto.

Por fim, utiliza-se o predicado

encomendasCamiao(Sorted,Encs,0,CargaMaxima).

para garantir que a soma de todas as encomendas escolhidas é igual ou inferior à capacidade máxima do camião.

De seguida, obtêm-se os pontos inicial e final, bem como a autonomia, sendo passados para um novo predicado, bem como a lista de encomendas obtida anteriormente:

calculaCaminho(Algoritmo,Ei,Encomendas,C,Autonomia).

Este predicado começa por chamar um predicado auxiliar

calculaCaminhoAux(Algoritmo,Ei,Encomendas,Resultado).

que faz o cálculo de todos os custos e caminhos para as encomendas possíveis, dado um ponto inicial, colocando numa lista no formato Encomenda-Custo-Caminho. Este cálculo varia mediante o valor passado em Algoritmo.

Com esta lista, chama-se o predicado

minimo(Resultado,Min,C,Custo).

que vai buscar qual é o caminho para a encomenda que possui menor custo, desde o ponto inicial. De seguida, é apagada a encomenda para a qual foi encontrado o melhor caminho, sendo o predicado **calculaCaminho** chamado novamente, mas agora o ponto inicial corresponde à encomenda que foi escolhida anteriormente, sendo todo este processo repetido para encontrar o melhor caminho desde essa encomenda até a uma nova futura encomenda.

É importante realçar que antes da chamada recursiva é efetuada a verificação se o camião possui autonomia suficiente para se deslocar desde o ponto onde se encontra até ao ponto seguinte. Se for possível, essa distância é deduzida na sua autonomia. Caso não seja possível, chama-se o seguinte predicado:

bombaMaisPerto(Algoritmo,Ei,Ef,CustoBomba).

Este vai colocar em Ef qual é a bomba mais próxima do ponto em que o camião se encontra (Ei), verificando se o camião possui autonomia para se deslocar até a essa mesma bomba, cuja distância equivale a CustoBomba.

Terminado o cálculo do caminho desde o ponto inicial até à última encomenda, falta ainda o camião deslocar-se desde a última encomenda até ao ponto final. Desta forma, vai-se buscar qual foi a última entrega e adiciona-se à lista de encomendas o valor do ponto final, chamando novamente o predicado calculaCaminho, agora com ponto inicial correspondente à última entrega, e ponto final correspondente.

Pode ocorrer de o programa indicar que encontrou um caminho impossível e continuar a correr. Se tal acontecer, quer dizer que está a tentar gerar outro caminho. Esta situação pode ocorrer várias vezes numa só execução.

4 Experiências

4.1 Objectivo de cada experiência

Foram gerados vários ficheiros de teste, com complexidades crescentes, usando o "DataGenerator". O nome dos ficheiros gerados é "dados*.pl", em que * é um número positivo que cresce conforme aumenta a complexidade. É importante referir que o grau máximo indicado ao gerador de dados nos seus argumentos foi sempre 1, pois corresponde a uma distância bastante grande para um número reduzido de pontos do grafo. Com isto, usámos pontos do grafo que ocupam apenas até 1º de longitude e 1º grau de latitude. Também fizemos pequenas alterações à autonomia e capacidade de carga do camião nos ficheiros de dados.

Com este ficheiros, foram realizados testes para determinar quanto tempo demoraria cada algoritmo a encontrar o caminho correspondente para cada percurso. Foi medido o tempo de execução dez vezes para cada algoritmo e foi feita a média desses valores, para uma medição mais precisa. Para os dois últimos conjuntos, não são apresentados os tempos de todos os algoritmos, pois o tempo de resolução desses algoritmos é muito elevado.

4.2 Resultados

Para um conjunto de 10 pontos do grafo e 2 encomendas, os tempos obtidos foram:

- pp(maxEntregas): 6.9ms
- pp(minDist): 7.7ms
- pl(maxEntregas): 8.2ms
- pl(minDist): 7.8ms
- a*(maxEntregas): 9.8ms
- a*(minDist): 9.8ms
- ida*(maxEntregas): 8.6ms
- ida*(minDist): 8.7ms

Para um conjunto de 20 pontos do grafo e 4 encomendas, os tempos obtidos foram:

- pp(maxEntregas): 54.5ms
- pp(minDist): 52.3ms
- pl(maxEntregas): 148.7ms
- pl(minDist): 153.2ms
- a*(maxEntregas): 102ms
- a*(minDist): 114.7ms
- ida*(maxEntregas): 150.5ms
- ida*(minDist): 114.7ms

Para um conjunto de 40 pontos do grafo e 8 encomendas, os tempos obtidos foram:

- pp(maxEntregas): 173.8ms
- pp(minDist): 196.4ms
- pl(maxEntregas): 73.6ms
- pl(minDist): 110.3ms
- ida*(maxEntregas): 6707ms
- ida*(minDist): 13414.7ms

Para um conjunto de 50 pontos do grafo e 15 encomendas, os tempos obtidos foram:

- pl(maxEntregas): 205.5ms
- pl(minDist): 428.8ms

5 Conclusões

Como já seria de prever, a experiência que utilizou um menor número de pontos e de encomendas obtém muito mais rapidamente a solução do que quando se dobram estes valores.

Analisando os tempos medidos nas duas primeiras experiências, é possível perceber que quando a complexidade do problema aumenta para o dobro, os tempos no algoritmo **"Primeiro em profundidade"** aumentam para cerca de 7 vezes mais, no **"Primeiro em largura"** aumentam para cerca de 18 vezes mais, no **"A*"** para cerca de 10 vezes mais e no **"IDA*"** para 15 vezes mais.

Os tempos das outras duas experiências foram medidos usando um processador mais rápido, não sendo possível usá-los para comparação. Para além disso, nem todos os algoritmos foram executados em tempo útil.

Uma vez que o algoritmo **"Primeiro em profundidade"** não utiliza qualquer tipo de informação e retorna logo o primeiro caminho encontrado, é normal este ser aquele que é mais rápido e que não aumenta tão significativamente com o aumento da dimensão da amostra.

No caso do **"Primeiro em largura"**, uma vez que percorre horizontalmente todo o grafo, é normal demorar mais que o anterior até encontrar uma solução, principalmente se esta não for tão direta.

O **"A*"**, já sendo um algoritmo de pesquisa informado, na primeira experiência verifica-se que é ligeiramente mais lento a dar um resultado comparativamente aos algoritmos anteriores, devido ao cálculo da heurística e de ter que armazenar toda a informação dos nós, apesar de oferecer um melhor resultado para o caminho. No entanto, este mostra-se melhor que o **"Primeiro em largura"** na segunda experiência, comprovando a ideia de que os métodos informados sejam melhores quando as amostras são maiores.

Quanto ao **"IDA*"**, esperava-se que fosse melhor que o **"A*"**, o que acontece na primeira experiência. No entanto, na segunda já se demonstrou pior, o que pode indicar que quando a amostra é grande não é tão eficaz ou que a árvore gerada na segunda experiência não é boa para o uso de pesquisa iterativa em profundidade (com aprofundamento limitado e incremental).

No conjunto com mais dados, pode-se verifcar que a ordem de pesquisa da solução é importante, já que foi possível correr o algoritmo em tempo útil realizando pesquisa em largura, mas tal não foi possível na pesquisa em profundidade. Tal deve estar relacionado com a solução estar num nível próximo da raiz, mas numa posição em relação aos irmãos muito afastada do 1º irmão, sendo que a profundidade da árvore gerada deve ser muito elevada.

6 Melhoramentos

No futuro, poder-se-iam realizar as seguintes melhorias:

- permitir o uso de mais do que um camião, calculando-se as encomendas e o percurso de um camião e usando as encomendas restantes para calcular o percurso de outros camiões;
- melhorar o algoritmo de decisão do caminho a seguir, sendo que, com isso, o seu tempo de execução também melhoraria;
- conseguir que a última encomenda a entregar estivesse o mais próximo possível do ponto final;
- mover o projecto para C++ de forma a melhorar consideravelmente o tempo de execução.

7 Recursos

7.1 Bibliografia

- Slides teóricos da unidade curricular, relativos aos "Métodos de Resolução de Problemas e Algoritmos para a Evolução"

7.2 Software

- SICStus Prolog VC14 4.3.3
- Notepad++
- Visual Studio Code
- Visual Studio 2017

7.3 Elementos do grupo

O grupo trabalhou sempre em conjunto, de forma a que cada elemento ficasse com uma carga igual e justa. Desta forma, pode-se afirmar que cada elemento tem uma percentagem aproximada de 33,3%.

A Apêndice

A.1 Manual do utilizador

Para utilizar o programa basta chamar um dos seguintes predicados:

- entregaEncomendas(pp, maxEntregas).
- entregaEncomendas(pp, minDist).
- entregaEncomendas(pl, maxEntregas).
- entregaEncomendas(pl, minDist).
- entregaEncomendas(astar, maxEntregas).
- entregaEncomendas(astar, minDist).
- entregaEncomendas(idastar, maxEntregas).
- entregaEncomendas(idastar, minDist).

O primeiro argumento do predicado corresponde ao algoritmo que se deseja utilizar: primeiro em profundidade (pp), primeiro em largura (pl), A* (astar) ou IDA* (idastar). Quanto ao segundo argumento, indica o objectivo, isto é, se deve maximizar o número de entregas (maxEntregas) ou minimizar a distância (minDist).